Contents

Dart Introduction & Basics

Dart Introduction & Basics

This section will help you to setup environment for dart development and learn the basics of dart. Here you will learn the following topics:

- Introduction to Dart,
- How to install Dart,
- Basic Dart Program,
- Variables in Dart,
- Datatypes in Dart,
- Comments in Dart,
- Operators in Dart,
- User Input in Dart, and
- String in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

Accelerate your Workflow

Save any snippet in this tutorial to your personal micro-repository with Pieces for Developers to speed up your workflow. Pieces is a centralized productivity suite that leverages AI to help developers save snippets, extract code from screenshots, auto-enrich code, and much more.

Dart

Dart is a client-optimized, object-oriented, modern programming language to build apps fast for many platforms like android, iOS, web, desktop, etc. Client optimized means optimized for crafting a beautiful user interface and high-quality experiences. Google developed Dart as a programming language.

Currently, Dart is one of the most preferred languages to learn. A solid understanding of Dart is necessary to develop high-quality apps with flutter. According to Github, **Dart is one of the most loved programming languages in the world.**

If you know languages like **C**, **Java**, **C#**, **Javascript**, etc. Dart will be easy for you. This tutorial covers Dart from basic to advance.

Note: The concept you will learn here will give you a solid understanding of Dart and improve your coding career.

Dart Features

- Free and open-source.
- Object-oriented programming language.
- Used to develop android, iOS, web, and desktop apps fast.
- Can compile to either native code or javascript.
- Offers modern programming features like null safety and asynchronous programming.
- You can even use Dart for servers and backend.

Difference Between Dart & Flutter

Dart

Dart is a client optimized, object-oriented programming language. It is popular nowadays because of flutter. It is difficult to build complete apps only using Dart because you have to manage many things yourself.

Flutter

Flutter is a framework that uses dart programming language. With the help of flutter, you can build apps for android, iOS, web, desktop, etc. The framework contains ready-made tools to make apps faster.

Which Is The Best Code Editor For Dart Programming

The best code editor is VS Code if you want to run the dart program from a computer or laptop. You can download the dart extension from VS Code and start coding. You will learn more about installing dart in the next topic. You can also use DartPad to run simple dart programs without installing anything.

Dart History

- Google developed Dart in 2011 as an alternative to javascript.
- Dart 1.0 was released on November 14, 2013.
- Dart 2.0 was released in August 2018.
- Dart 3.0 was released in May 2023.
- Dart gained popularity in recent days because of flutter.

Basic Programming Terms

Important words that you often hear while learning programming languages.

Statements:

A statement is a command that tells a computer to do something. In Dart, you can end most statements with a semicolon **;**.

Expressions:

An Expression is a value or something that can be calculated as a value. The expression can be numbers, text, or some other type. For E.g.

```
a. 52
b. 5+5
c. 'Hello World.'
d. num
```

Keywords:

Keywords are reserved words that give special meaning to the dart compiler. For E.g. **int**, **if**, **var**, **String**, **const**, etc.

Identifiers:

Identifiers are names created by the programmer to define variables, functions, classes, etc. Identifiers shouldn't be keywords and must have a unique name. For E.g. **int age =19;**, here age is an identifier. You will learn more about identifiers later in this course.

High-Level Programming Language:

High-Level Programming Language is easy to learn, user-friendly, and uses English-like-sentence. For E.g. dart,c,java,etc.

Low-Level Programming Language:

Low-level programming language is hard to learn, non-user friendly, and deals with computer hardware components, e.g., machine and assembly language.

Note: Low-level languages are faster than high-level but hard to understand and debug.

Compiler:

A compiler is a computer program that translates the high-level programming language into machine-level language.

Syntax:

The Syntax is a programming language's pattern or rules that give the concept to code.

Key Points

- Dart is a free and open-source programming language. You don't need to pay any money to run dart programs.
- Dart is a platform-independent language and supports almost every operating system such as windows, mac, and Linux.
- Dart is an object-oriented programming language and supports all oops features such as encapsulation, inheritance, polymorphism, interface, etc.
- Dart comes with a **dart2js** compiler which translates dart code to javascript code that runs on all modern browsers.
- Dart is a programming language used by flutter, the world's most popular framework for building apps.

INSTALL DART

Dart Installation

There are multiple ways to install a dart on your system. You can install Dart on **Windows, Mac, and Linux** or run it from the browser.

Requirements

- **Dart SDK**,
- **VS code or other editors** like Intellij [We will use VS Code here].

Dart Windows Installation

Follow the below instructions to install a dart on the windows operating system.

Steps:

- Download Dart SDK from here.
- Copy **dart-sdk** folder to your C drive.
- Add **C:\dart-sdk\bin** to your environment variable. Watch the video below to be more clear.
- Open the command prompt and type `dart --version` to check it.
- Install VS Code and Add Dart Extension.

Note: Dart SDK provides the tools to compile and run dart program.

Dart Mac Installation

- Install Homebrew From here.

- Type `brew tap dart-lang/dart` in the terminal.
- Type `brew install dart` in the terminal.
- If you have any issues installing the dart, watch the video below.

Homebrew Install Command

Copy and paste this command on your terminal to install Homebrew.

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

To set the homebrew path, copy and paste this command on your terminal.

```
export PATH=/opt/homebrew/bin:$PATH
```

Dart Linux Installation

To install a dart on Linux, open your terminal and **copy/paste** the below commands.

```
sudo apt-get update
sudo apt-get install apt-transport-https
wget -qO- https://dl-ssl.google.com/linux/linux_signing_key.pub | sudo gpg
--dearmor -o /usr/share/keyrings/dart.gpg
echo 'deb [signed-by=/usr/share/keyrings/dart.gpg arch=amd64]
https://storage.googleapis.com/download.dartlang.org/linux/debian stable
main' | sudo tee /etc/apt/sources.list.d/dart_stable.list
```

Then, install the dart using the below command.

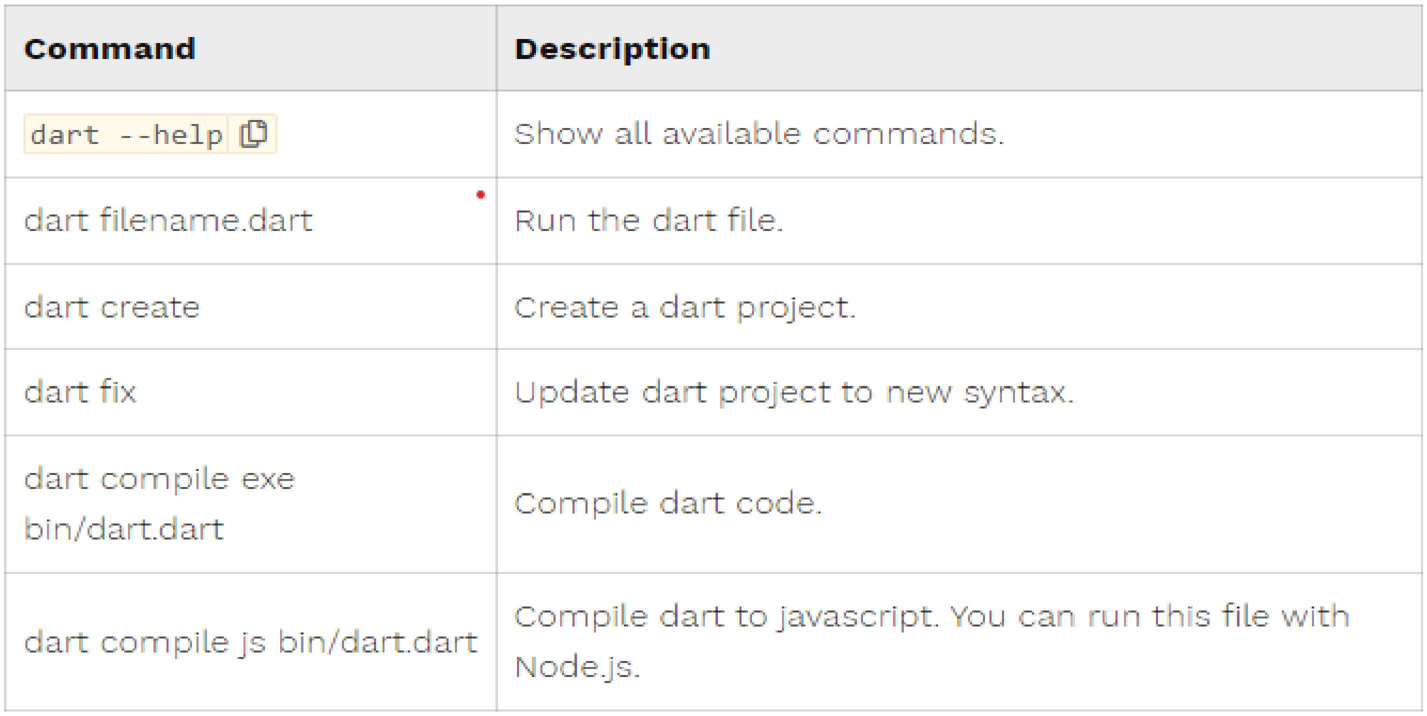
```
sudo apt-get update
sudo apt-get install dart
```

To set the dart path, copy and paste this command on your terminal.

```
export PATH="$PATH:/usr/lib/dart/bin"
```

Check Dart Installation

Open your command prompt and type **`dart --version`**. The dart is successfully installed on your system if it gives you a version code. If not, watch the video above.



Command	Description
`dart --help`	Show all available commands.
dart filename.dart	Run the dart file.
dart create	Create a dart project.
dart fix	Update dart project to new syntax.
dart compile exe bin/dart.dart	Compile dart code.
dart compile js bin/dart.dart	Compile dart to javascript. You can run this file with Node.js.

Some Useful Commands

Run Dart On Web

You can run the dart program on your browser without installing any software. Dartpad is a web tool to write and run your dart code.

- Run Dart Programming on Web

Install Dart Official Link

- Install Dart Official Link

Can You Run Dart From Mobile?

Yes, you can use DartPad to run simple dart programs from your phone without installing any software. For bigger projects, using DartPad is not recommended.

BASIC DART PROGRAM

Basic Dart Program

This is a simple dart program that prints **Hello World** on screen. Most programmers write the Hello World program as their first program.

```
void main() {
   print("Hello World!");
}
```

Show Output

```
Hello World!
```

Basic Dart Program Explained

- void main() is the starting point where the execution of your program begins.
- Every program starts with a main function.
- The curly braces {} represent the beginning and the ending of a block of code.
- print(“Hello World!”); prints Hello World! on screen.
- Each code statement must end with a semicolon.

Basic Dart Program For Printing Name

```
void main()
{
    var name = "John";
    print(name);
}
```

Show Output

```
John
```

Dart Program To Join One Or More Variables

Here **$variableName** is used to join variables. This joining process in dart is called string interpolation.

```
void main(){
  var firstName = "John";
  var lastName = "Doe";
  print("Full name is $firstName $lastName");
}
```

Show Output

```
Full name is John Doe
```

Dart Program For Basic Calculation

Performing addition, subtraction, multiplication, and division in dart.

```
void main() {
int num1 = 10; //declaring number1
int num2 = 3; //declaring number2

// Calculation
int sum = num1 + num2;
int diff = num1 - num2;
int mul = num1 * num2;
double div = num1 / num2; // It is double because it outputs number with decimal.

// displaying the output
print("The sum is $sum");
print("The diff is $diff");
print("The mul is $mul");
print("The div is $div");
}
```

Show Output

```
The sum is 13
The diff is 7
The mul is 30
The div is 3.3333333333333335
```

Create Full Dart Project

It's nice to work on a single file, but if your project gets bigger, you need to manage configurations, packages, and assets files. So creating a dart project will help you to manage this all.

```
dart create <project_name>
```

This will create a simple dart project with some ready-made code.

Steps To Create Dart Project

- Open folder location on command prompt/terminal.
- Type `dart create project_name` (For E.g. dart create first_app)
- Type `cd first_app`
- Type `code .` to open project with visual studio code
- To check the main dart file go to **bin/first_app.dart** and edit your code.

Run Dart Project

First, open the project location on the command/terminal and run the project with this command.

```
dart run
```

Command	Description
dart compile js filename.dart	Compile dart to javascript. You can run this file with Node.js.

Challenge

Create a new dart project with name **stockmanagement** and then run it.

VARIABLES IN DART

Variables

Variables are containers used to store value in the program. There are different types of variables where you can keep different kinds of values. Here is an example of creating a variable and initializing it.

```
// here variable name contains value John.
var name = "John";
```

Variable Types

They are called data types. We will learn more about data types later in this dart tutorial.

- **String**: For storing text value. E.g. “John” [Must be in quotes]
- **int**: For storing integer value. E.g. 10, -10, 8555 [Decimal is not included]
- **double**: For storing floating point values. E.g. 10.0, -10.2, 85.698 [Decimal is included]
- **num**: For storing any type of number. E.g. 10, 20.2, -20 [both int and double]
- **bool**: For storing true or false. E.g. true, false [Only stores true or false values]
- **var**: For storing any value. E.g. ‘Bimal’, 12, ‘z’, true

Syntax

This is syntax for creating a variable in dart.

```
type variableName = value;
```

Example 1: Using Variables In Dart

In this example, you will learn how to declare variables and print their values.

```
void main() {
```

```
// declaring variables
String name = "John";
String address = "USA";
num age = 20; // used to store any types of numbers
num height = 5.9;
bool isMarried = false;

// printing variables value
print("Name is $name");
print("Address is $address");
print("Age is $age");
print("Height is $height");
print("Married Status is $isMarried");
}
```
Show Output

```
Name is John
Address is USA
Age is 20
Height is 5.9
Married Status is false
```

Note: Always use the descriptive variable name. Don't use a variable name like a, b, c because this will make your code more complex.

Rules For Creating Variables In Dart

- Variable names are case sensitive, i.e., a and A are different.
- A variable name can consist of letters and alphabets.
- A variable name cannot start with a number.
- Keywords are not allowed to be used as a variable name.
- Blank spaces are not allowed in a variable name.
- Special characters are not allowed except for the underscore (_) and the dollar ($) sign.

Dart Constant

Constant is the type of variable whose value never changes. In programming, changeable values are **mutable** and unchangeable values are **immutable**. Sometimes, you don't need to change the value once declared. Like the value of PI=3.14, it never changes. To create a constant in Dart, you can use the const keyword.

```
void main(){
const pi = 3.14;
pi = 4.23; // not possible
print("Value of PI is $pi");
}
```
Show Output

```
Constant variables can't be assigned a value.
```

Naming Convention For Variables In Dart

It is a good habit to follow the naming convention. In Dart Variables, the variable name should start with lower-case, and every second word's first letter will be upper-case like num1, fullName, isMarried, etc. Technically, this naming convention is called **lowerCamelCase**.

Naming Convention Example

```
// Not standard way
var fullname = "John Doe";
// Standard way
var fullName = "John Doe";
const pi = 3.14;
```

DATA TYPES IN DART

Data Types

Data types help you to categorize all the different types of data you use in your code. **For e.g. numbers, texts, symbols, etc**. The data type specifies what type of value will be stored by the variable. Each variable has its data type. Dart supports the following built-in data types :

1. Numbers
2. Strings
3. Booleans
4. Lists
5. Maps
6. Sets
7. Runes
8. Null

In Dart language, there is the type of values that can be represented and manipulated. The data type classification is as given below:

Data Type	Keyword	Description
Numbers	int, double, num	It represents numeric values
Strings	String	It represents a sequence of characters
Booleans	bool	It represents Boolean values true and false
Lists	List	It is an ordered group of items
Maps	Map	It represents a set of values as key-value pairs
Sets	Set	It is an unordered list of unique values of same types
Runes	runes	It represents Unicode values of String
Null	null	It represents null value

Numbers

When you need to store numeric value on dart, you can use either int or double. Both int and double are subtypes of **num**. You can use num to store both int or double value.

```
void main() {
// Declaring Variables
int num1 = 100; // without decimal point.
double num2 = 130.2; // with decimal point.
num num3 = 50;
num  num4 = 50.4;

// For Sum
num sum = num1 + num2 + num3 + num4;

// Printing Info
print("Num 1 is $num1");
print("Num 2 is $num2");
print("Num 3 is $num3");
print("Num 4 is $num4");
print("Sum is $sum");

}
```

Show Output

```
Num 1 is 100
Num 2 is 130.2
Num 3 is 50
Num 4 is 50.4
Sum is 330.59999999999997
```

Round Double Value To 2 Decimal Places

The `.toStringAsFixed(2)` is used to round the double value upto 2 decimal places in dart. You can round to any decimal places by entering numbers like 2, 3, 4, etc.

```
void main() {
// Declaring Variables
double price = 1130.2232323233233; // valid.
print(price.toStringAsFixed(2));
}
```

Show Output

```
1130.22
```

String

String helps you to store text data. You can store values like **I love dart**, **New York 2140** in String. You can use single or double quotes to store string in dart.

```
void main() {
// Declaring Values
String schoolName = "Diamond School";
String address = "New York 2140";

// Printing Values
print("School name is $schoolName and address is $address");
}
```

Show Output

```
School name is Diamond School and address is New York 2140
```

Create A Multi-Line String In Dart

If you want to create a multi-line String in dart, then you can use triple quotes with either single or double quotation marks.

```
void main() {
// Multi Line Using Single Quotes
String multiLineText = '''
This is Multi Line Text
with 3 single quote
I am also writing here.
''';
// Multi Line Using Double Quotes
String otherMultiLineText = """
This is Multi Line Text
with 3 double quote
I am also writing here.
""";

// Printing Information
print("Multiline text is $multiLineText");
print("Other multiline text is $otherMultiLineText");
}
```

Show Output

```
Multiline text is This is Multi Line Text
with 3 single quote
I am also writing here.

Other multiline text is This is Multi Line Text
with 3 double quote
I am also writing here.
```

Special Character In String

Special Character	**Work**
\n	New Line
\t	Tab

```
void main() {

// Using \n and \t
print("I am from \nUS.");
print("I am from \tUS.");
}
```

Show Output

```
I am from
US.
I am from      US.
```

Create A Raw String In Dart

You can also create raw string in dart. Special characters won't work here. You must write **r** after equal sign.

```
void main() {
// Set price value
num price = 10;
String withoutRawString = "The value of price is \t $price"; // regular
String
String withRawString =r"The value of price is \t $price"; // raw String
print("Without Raw: $withoutRawString"); // regular result
print("With Raw: $withRawString"); // with raw result

}
```
Show Output
```
Without Raw: The value of price is    10
With Raw: The value of price is \t $price
```
Type Conversion In Dart

In dart, type conversion allows you to convert one data type to another type. For e.g. to convert String to int, int to String or String to bool, etc.

Convert String To Int In Dart

You can convert String to int using int.parse() method. The method takes String as an argument and converts it into an integer.

```
void main() {
String strvalue = "1";
print("Type of strvalue is ${strvalue.runtimeType}");
int intvalue = int.parse(strvalue);
print("Value of intvalue is $intvalue");
// this will print data type
print("Type of intvalue is ${intvalue.runtimeType}");
}
```
Show Output
```
Type of strvalue is String
Value of intvalue is 1
```
Type of intvalue is int

Convert String To Double In Dart

You can convert String to double using double.parse() method. The method takes String as an argument and converts it into a double.

```
void main() {
String strvalue = "1.1";
print("Type of strvalue is ${strvalue.runtimeType}");
double doublevalue = double.parse(strvalue);
print("Value of doublevalue is $doublevalue");
// this will print data type
print("Type of doublevalue is ${doublevalue.runtimeType}");
}
```
Show Output
```
Type of strvalue is String
Value of doublevalue is 1.1
```

```
Type of doublevalue is double
```

Convert Int To String In Dart

You can convert int to String using the toString() method. Here is example:

```
void main() {
int one = 1;
print("Type of one is ${one.runtimeType}");
String oneInString = one.toString();
print("Value of oneInString is $oneInString");
// this will print data type
print("Type of oneInString is ${oneInString.runtimeType}");
}
```

Show Output

```
Type of one is int
Value of oneInString is 1
Type of oneInString is String
```

Convert Double To Int In Dart

You can convert double to int using the toInt() method.

```
void main() {
   double num1 = 10.01;
   int num2 = num1.toInt(); // converting double to int

  print("The value of num1 is $num1. Its type is ${num1.runtimeType}");
  print("The value of num2 is $num2. Its type is ${num2.runtimeType}");
}
```

Show Output

```
The value of num1 is 10.01. Its type is double
The value of num2 is 10. Its type is int
```

Booleans

In Dart, boolean holds either true or false value. You can write the **bool** keyword to define the boolean data type. You can use boolean if the answer is true or false. Consider the answer to the following questions:

- Are you married?
- Is the door open?
- Does a cat fly?
- Is the traffic light green?
- Are you older than your father?

These all are yes/no questions. Its a good idea to store them in boolean.

```
void main() {
```

```
bool isMarried = true;
print("Married Status: $isMarried");
}
```

Show Output

```
Married Status: true
```

Lists

The list holds multiple values in a single variable. It is also called arrays. If you want to store multiple values without creating multiple variables, you can use a list.

```
void main() {
List<String> names = ["Raj", "John", "Max"];
print("Value of names is $names");
print("Value of names[0] is ${names[0]}"); // index 0
print("Value of names[1] is ${names[1]}"); // index 1
print("Value of names[2] is ${names[2]}"); // index 2

  // Finding Length of List
int length = names.length;
print("The Length of names is $length");
}
```

Show Output

```
Value of names is [Raj, John, Max]
Value of names[0] is Raj
Value of names[1] is John
Value of names[2] is Max
The Length of names is 3
```

Note: List index always starts with 0. Here names[0] is Raj, names[1] is John and names[2] is Max.

Sets

An unordered collection of unique items is called set in dart. You can store unique data in sets.

Note: Set doesn’t print duplicate items.

```
void main() {
Set<String> weekday = {"Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"};
print(weekday);
}
```

Show Output

```
{Sun, Mon, Tue, Wed, Thu, Fri, Sat}
```

Maps

In Dart, a map is an object where you can store data in key-value pairs. Each key occurs only once, but you can use same value multiple times.

```
void main() {
Map<String, String> myDetails = {
   'name': 'John Doe',
   'address': 'USA',
   'fathername': 'Soe Doe'
};
// displaying the output
print(myDetails['name']);
}
```

Show Output

```
John Doe
```

Var Keyword In Dart

In Dart, **var** automatically finds a data type. In simple terms, var says if you don't want to specify a data type, I will find a data type for you.

```
void main(){

var name = "John Doe"; // String
var age = 20; // int

print(name);
print(age);
}
```

Show Output

```
John Doe
20
```

Runes In Dart

With runes, you can find Unicode values of String. The Unicode value of **a** is **97**, so runes give 97 as output.

```
void main() {

String value = "a";
print(value.runes);
}
```

Show Output

```
(97)
```

How To Check Runtime Type

You can check runtime type in dart with `.runtimeType` after the variable name.

```
void main() {
   var a = 10;
   print(a.runtimeType);
   print(a is int); // true
}
```

Show Output

```
int
true
```

Optionally Typed Language

You may have heard of the **statically-typed** language. It means the data type of variables is known at compile time. Similarly, **dynamically-typed** language means data types of variables are known at run time. Dart supports dynamic and static types, so it is called optionally-typed language.

Statically Typed

A language is statically typed if the data type of variables is known at compile time. Its main advantage is that the compiler can quickly check the issues and detect bugs.

```
void main() {
   var myVariable = 50; // You can also use int instead of var
   myVariable = "Hello"; // this will give error
   print(myVariable);
}
```

Show Output

```
Error:
A value of type 'String' can't be assigned to a variable of type 'int'.
```

Dynamically Typed Example

A language is dynamically typed if the data type of variables is known at run time.

```
void main() {
   dynamic myVariable = 50;
   myVariable = "Hello";
   print(myVariable);
}
```

Show Output

```
Hello
```

Note: Using static type helps you to prevent writing silly mistakes in code. It's a good habit to use static type in dart.

COMMENTS IN DART

Comments

Comments are the set of statements that are ignored by the dart compiler during program execution. They are used to explain the code so that you or other people can understand it easily.

Advantages Of Comments

- You can describe your code.
- Other people will understand your code more clearly.

Types Of Comments

- **Single-Line Comment**: For commenting on a single line of code. E.g. // This is a single-line comment.
- **Multi-Line Comment**: For commenting on multiple lines of code. E.g. /* This is a multi-line comment. */
- **Documentation Comment**: For generating documentation or reference for a project/software package. E.g. /// This is a documentation comment

Single-Line Comment In Dart

Single line comments start with `//` in dart. You can write `//` and your text.

```
void main() {
// This is single-line comment.
print("Welcome to Technology Channel.");
}
```

Show Output

```
Welcome to Technology Channel.
```

Multi-Line Comment In Dart

Multi-line comments start with `/*` and end with `*/` . You can write your comment inside `/*` and `*/`.

```
void main(){
/*
This is a multi-line comment.
*/
    print("Welcome to Technology Channel.");
}
```

Show Output

```
Welcome to Technology Channel.
```

Documentation Comment In Dart

Documentation comments are helpful when you are writing documentation for your code. Documentation comments start with `///` in dart.

```
void main(){
/// This is documentation comment
    print("Welcome to Technology Channel.");
}
```

Show Output

```
Welcome to Technology Channel.
```

OPERATORS IN DART

Operators In Dart

Operators are used to perform mathematical and logical operations on the variables. Each operation in dart uses a symbol called the operator to denote the type of operation it performs. Before learning operators in the dart, you must understand the following things.

- **Operands** : It represents the data.
- **Operator** : It represents how the operands will be processed to produce a value.

Note: Suppose the given expression is 2 + 3. Here 2 and 3 are operands, and `+` is the operator.

Types Of Operators

There are different types of operators in dart. They are as follows:

- **Arithmetic Operators**
- **Increment and Decrement Operators**
- **Assignment Operators**
- **Logical Operators**
- **Type Test Operators**

Arithmetic Operators

Arithmetic operators are the most common types of operators. They perform operations like addition, subtraction, multiplication, division, etc.

Operator Symbol	**Operator Name**	**Description**
`+`	Addition	For adding two operands
`-`	Subtraction	For subtracting two operands
`-expr`	Unary Minus	For reversing the sign of the expression
`*`	Multiplication	For multiplying two operands
`/`	Division	For dividing two operands and give output in double
`~/`	Integer Division	For dividing two operands and give output in integer
`%`	Modulus	Remainder After Integer Division

Let's look at how to perform arithmetic calculations in dart.

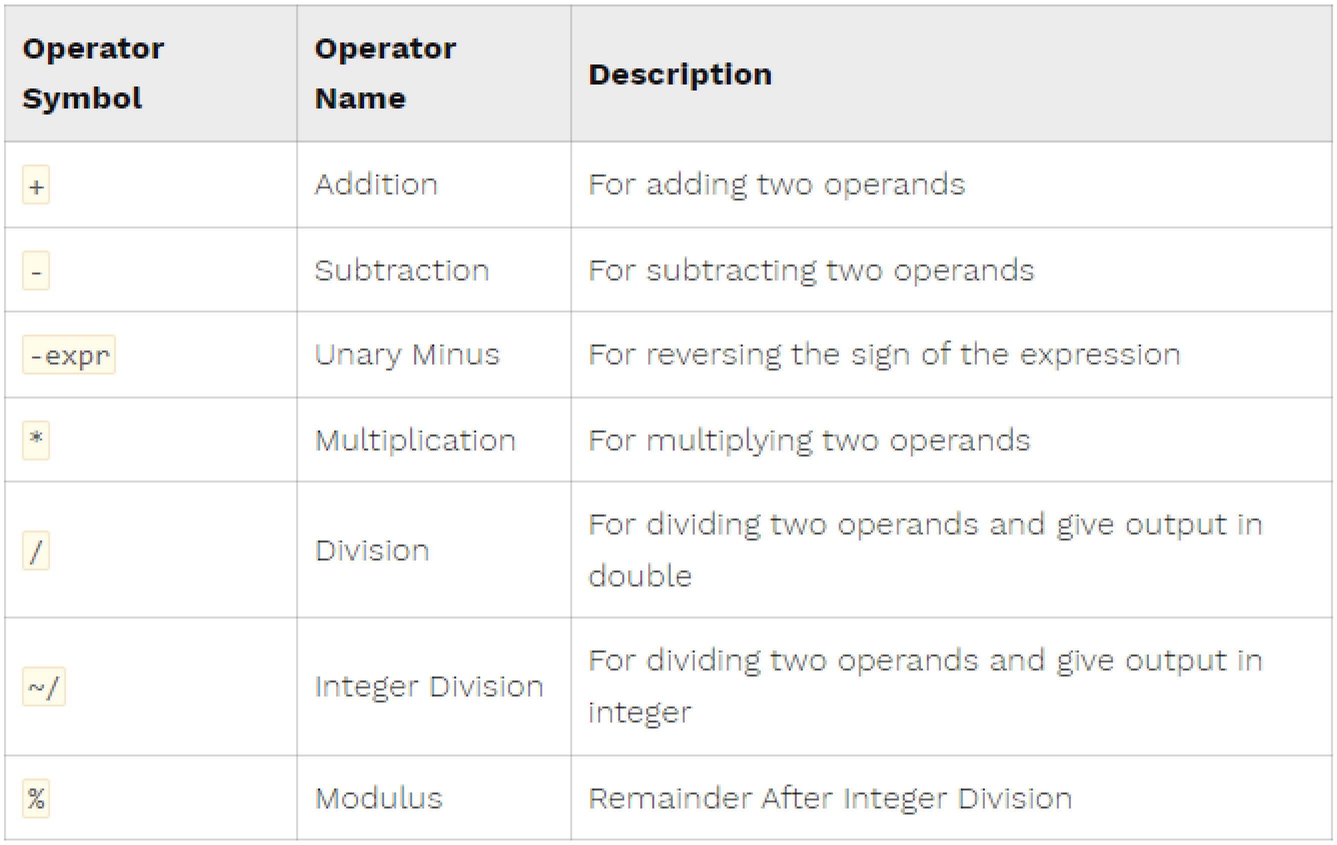
```
void main() {
 // declaring two numbers
 int num1=10;
 int num2=3;

 // performing arithmetic calculation
 int sum=num1+num2;        // addition
 int diff=num1-num2;       // subtraction
 int unaryMinus = -num1;     // unary minus
 int mul=num1*num2;        // multiplication
 double div=num1/num2;     // division
 int div2 =num1~/num2;      // integer division
 int mod=num1%num2;        // show remainder

//Printing info
 print("The addition is $sum.");
 print("The subtraction is $diff.");
 print("The unary minus is $unaryMinus.");
 print("The multiplication is $mul.");
 print("The division is $div.");
 print("The integer division is $div2.");
 print("The modulus is $mod.");
}
```

Show Output

```
The addition is 13.
The subtraction is 7.
The unary minus is -10.
The multiplication is 30.
The division is 3.3333333333333335.
The integer division is 3.
The modulus is 1.
```

Increment and Decrement Operators

With increment and decrement operators, you can increase and decrease values. If ++ is used at the beginning, then it is a prefix. If it is used at last, then it is postfix.



Operator Symbol	Operator Name	Description
`+`	Addition	For adding two operands
`-`	Subtraction	For subtracting two operands
`-expr`	Unary Minus	For reversing the sign of the expression
`*`	Multiplication	For multiplying two operands
`/`	Division	For dividing two operands and give output in double
`~/`	Integer Division	For dividing two operands and give output in integer
`%`	Modulus	Remainder After Integer Division

Note: ++var increases the value of operands, whereas var++ returns the actual value of operands before the increment.

```
void main() {
// declaring two numbers
 int num1=0;
 int num2=0;

// performing increment / decrement operator

// pre increment
num2 = ++num1;
print("The value of num2 is $num2");

// reset value to 0
num1 = 0;
num2 = 0;

// post increment
num2 =  num1++;
print("The value of num2 is $num2");

}
```

Show Output

```
The value of num2 is 1
The value of num2 is 0
```

Assignment Operators

It is used to assign some values to variables. Here, we are assigning 24 to the age variable.

```
int age = 24;
```

Operator Type	**Description**
`=`	Assign a value to a variable
`+=`	Adds a value to a variable
`-=`	Reduces a value to a variable
`*=`	Multiply value to a variable
`/=`	Divided value by a variable

```dart
void main() {
  double age = 24;
  age+= 1;  // Here age+=1 means age = age + 1.
  print("After Addition Age is $age");
  age-= 1;  //Here age-=1 means age = age - 1.
  print("After Subtraction Age is $age");
  age*= 2;  //Here age*=2 means age = age * 2.
  print("After Multiplication Age is $age");
  age/= 2;  //Here age/=2 means age = age / 2.
  print("After Division Age is $age");
}
```

Show Output

```
After Addition Age is 25.0
After Aubtraction Age is 24.0
After Multiplication Age is 48.0
After Division Age is 24.0
```

Relational operators are also called comparison operators. They are used to make a comparison.

Operator Symbol	Operator Name	Description
`>`	Greater than	Used to check which operand is bigger and gives result as boolean
`<`	Less than	Used to check which operand is smaller and gives result as boolean
`>=`	Greater than or equal to	Used to check which operand is bigger or equal and gives result as boolean
`<=`	Less than or equal to	Used to check which operand is smaller or equal and gives result as boolean
`==`	Equal to	Used to check operands are equal to each other and gives result as boolean
`!=`	Not equal to	Used to check operand are not equal to each other and gives result as boolean

```
void main() {

 int num1=10;
 int num2=5;
 //printing info
 print(num1==num2);
 print(num1<num2);
 print(num1>num2);
 print(num1<=num2);
 print(num1>=num2);
}
```

Show Output

```
false
false
true
false
true
```

It is used to compare values.

Operator Type	Description
`&&`	This is 'and', return true if all conditions are true
`||`	This is 'or'. Return true if one of the conditions is true
`!`	This is 'not'. return false if the result is true and vice versa

```
void main(){
  int userid = 123;
    int userpin = 456;

    // Printing Info
    print((userid == 123) && (userpin== 456)); // print true
    print((userid == 1213) && (userpin== 456)); // print false.
    print((userid == 123) || (userpin== 456)); // print true.
    print((userid == 1213) || (userpin== 456)); // print true
    print((userid == 123) != (userpin== 456));//print false

}
```

Show Output

```
true
false
true
true
false
```

Type Test Operators

In Dart, type test operators are useful for checking types at runtime.

Operator Symbol	Operator Name	Description
`is`	is	Gives boolean value true if the object has a specific type
`is!`	is not	Gives boolean value false if the object has a specific type

```
void main() {
  String value1 = "Dart Tutorial";
  int age = 10;

  print(value1 is String);
  print(age is !int);
}
```

Show Output

```
true
false
```

USER INPUT IN DART

User Input In Dart

Instead of writing hard-coded values, you can give input to the computer. It will make your program more dynamic. You must import the package `import 'dart:io';` for user input.

Info

Note: You won't be able to take input from users using dartpad. You need to run a program from your computer.

String User Input

They are used for storing textual user input. If you want to keep values like somebody's name, address, description, etc., you can take string input from the user.

```
import 'dart:io';

void main() {
  print("Enter name:");
  String? name  = stdin.readLineSync();
  print("The entered name is ${name}");
}
```
Show Output

```
Enter name:
Raj Sharma
The entered name is Raj Sharma
```

Integer User Input

You can take integer input to get a numeric value from the user without the decimal point. E.g. 10, 100, -800 etc.

```
import 'dart:io';

void main() {
  print("Enter number:");
  int? number = int.parse(stdin.readLineSync()!);
  print("The entered number is ${number}");
}
```
Show Output

```
Enter number:
50
The entered number is 50
```

Floating Point User Input

You can use float input if you want to get a numeric value from the user with the decimal point. E.g. 10.5, 100.5, -800.9 etc.

```
import 'dart:io';

void main() {
  print("Enter a floating number:");
  double number = double.parse(stdin.readLineSync()!);
  print("The entered num is $number");
}
```
Show Output

```
Enter a floating number:
55.5
The entered num is 55.5
```

STRING IN DART

String In Dart

String helps you to store text based data. In String, you can represent your name, address, or complete book. It holds a series or sequence of characters – letters, numbers, and special characters. You can use single or double, or triple quotes to represent String.

Example: String In Dart

Single line String is written in single or double quotes, whereas multi-line strings are written in triple quotes. Here is an example of it:

```
void main() {
   String text1 = 'This is an example of a single-line string.';
   String text2 = "This is an example of a single line string using double
quotes.";
   String text3 = """This is a multiline line
string using the triple-quotes.
This is tutorial on dart strings.
""";
   print(text1);
   print(text2);
   print(text3);
}
```

Show Output

```
This is an example of a single-line string.
This is an example of a single line string using double quotes.
This is a multiline line
string using the triple-quotes.
This is tutorial on dart strings.
```

String Concatenation

You can combine one String with another string. This is called concatenation. In Dart, you can use the `+` operator or use **interpolation** to concatenate the String. Interpolation makes it easy to read and understand the code.

String Concatenation In Dart

```
void main() {
String firstName = "John";
String lastName = "Doe";
print("Using +, Full Name is "+firstName + " " + lastName+".");
print("Using interpolation, full name is $firstName $lastName.");

}
```

Show Output

```
Using +, Full Name is John Doe.
Using interpolation, full name is John Doe.
```

Properties Of String

- **codeUnits**: Returns an unmodifiable list of the UTF-16 code units of this string.
- **isEmpty**: Returns true if this string is empty.
- **isNotEmpty**: Returns false if this string is empty.
- **length**: Returns the length of the string including space, tab, and newline characters.

String Properties Example In Dart

```
void main() {
   String str = "Hi";
   print(str.codeUnits);   //Example of code units
   print(str.isEmpty);     //Example of isEmpty
   print(str.isNotEmpty);  //Example of isNotEmpty
   print("The length of the string is: ${str.length}");   //Example of
Length
}
```

Show Output

```
[72, 105]
false
true
The length of the String is: 2
```

Methods Of String

- **toLowerCase()**: Converts all characters in this string to lowercase.
- **toUpperCase()**: Converts all characters in this string to uppercase.
- **trim()**: Returns the string without any leading and trailing whitespace.
- **compareTo()**: Compares this object to another.
- **replaceAll()**: Replaces all substrings that match the specified pattern with a given value.
- **split()**: Splits the string at matches of the specified delimiter and returns a list of substrings.
- **toString()**: Returns a string representation of this object.
- **substring()**: Returns the text from any position you want.
- **codeUnitAt()**: Returns the 16-bit UTF-16 code unit at the given index.

String Methods Example In Dart

Here you will see various string methods that can help your work a lot better and faster.

Converting String To Uppercase and Lowercase

You can convert your text to lower case using .toLowerCase() and convert to uppercase using .toUpperCase() method.

```
//Example of toUpperCase() and toLowerCase()
void main() {
   String address1 = "Florida"; // Here F is capital
   String address2 = "TexAs"; // Here T and A are capital
   print("Address 1 in uppercase: ${address1.toUpperCase()}");
   print("Address 1 in lowercase: ${address1.toLowerCase()}");
   print("Address 2 in uppercase: ${address2.toUpperCase()}");
   print("Address 2 in lowercase: ${address2.toLowerCase()}");
}
```

Show Output

```
Address 1 in uppercase: FLORIDA
Address 1 in lowercase: florida
Address 2 in uppercase: TEXAS
Address 2 in lowercase: texas
```

Trim String In Dart

Trim is helpful when removing leading and trailing spaces from the text. This trim method will remove all the starting and ending spaces from the text. You can also use **trimLeft()** and **trimRight()** methods to remove space from left and right, respectively.

Note: The trim() method in Dart doesn’t remove spaces in the middle.

```
//Example of trim()
void main() {
  String address1 = " USA"; // Contain space at leading.
  String address2 = "Japan  "; // Contain space at trailing.
  String address3 = "New Delhi"; // Contains space at middle.

  print("Result of address1 trim is ${address1.trim()}");
  print("Result of address2 trim is ${address2.trim()}");
  print("Result of address3 trim is ${address3.trim()}");
  print("Result of address1 trimLeft is ${address1.trimLeft()}");
  print("Result of address2 trimRight is ${address2.trimRight()}");
}
```

Show Output

```
Result of address1 trim is USA
Result of address2 trim is Japan
Result of address3 trim is New Delhi
Result of address1 trimLeft is USA
Result of address2 trimRight is Japan
```

Compare String In Dart

In Dart, you can compare two strings. It will give the result 0 when two texts are equal, 1 when the first String is greater than the second, and -1 when the first String is smaller than the second.

```
//Example of compareTo()
void main() {
   String item1 = "Apple";
   String item2 = "Ant";
   String item3 = "Basket";

   print("Comparing item 1 with item 2: ${item1.compareTo(item2)}");
   print("Comparing item 1 with item 3: ${item1.compareTo(item3)}");
   print("Comparing item 3 with item 2: ${item3.compareTo(item2)}");
}
```

Show Output

```
Comparing item 1 with item 2: 1
Comparing item 1 with item 3: -1
Comparing item 3 with item 2: 1
```

Replace String In Dart

You can replace one value with another with the replaceAll(“old”, “new”) method in Dart. It will replace all the “old” words with “new”. Here in this example, this will replace milk with water.

```
//Example of replaceAll()
void main() {
String text = "I am a good boy I like milk. Doctor says milk is good for
health.";

String newText = text.replaceAll("milk", "water");

print("Original Text: $text");
print("Replaced Text: $newText");

}
```

Show Output

```
Original Text: I am a good boy I like milk. Doctor says milk is good for
health.
Replaced Text: I am a good boy I like water. Doctor says water is good for
health.
```

Split String In Dart

You can use the dart split method if you want to split String by comma, space, or other text. It will help you to split String to list.

```
//Example of split()
void main() {
  String allNames = "Ram, Hari, Shyam, Gopal";

  List<String> listNames = allNames.split(",");
  print("Value of listName is $listNames");

  print("List name at 0 index ${listNames[0]}");
  print("List name at 1 index ${listNames[1]}");
  print("List name at 2 index ${listNames[2]}");
  print("List name at 3 index ${listNames[3]}");

}
```

Show Output

```
Value of listName is [Ram,  Hari,  Shyam,  Gopal]
List name at 0 index Ram
List name at 1 index  Hari
List name at 2 index  Shyam
List name at 3 index  Gopal
```

ToString In Dart

In dart, toString() represents String representation of the value/object.

```
//Example of toString()
void main() {
int number = 20;
String result = number.toString();

print("Type of number is ${number.runtimeType}");
print("Type of result is ${result.runtimeType}");

}
```

Show Output

```
Type of number is int
Type of result is String
```

SubString In Dart

You can use substring in Dart when you want to get a text from any position.

```
//Example of substring()
void main() {
   String text = "I love computer";
   print("Print only computer: ${text.substring(7)}"); // from index 6 to
the last index
   print("Print only love: ${text.substring(2,6)}");// from index 2 to the
6th index
}
```

Show Output

```
Print only computer: computer
Print only love: love
```

Reverse String In Dart

If you want to reverse a String in Dart, you can reverse it using a different solution. One solution is here.

```
void main() {
  String input = "Hello";
  print("$input Reverse is ${input.split('').reversed.join()}");
}
```

Show Output

```
Hello Reverse is olleH
```

How To Capitalize First Letter Of String In Dart

If you want to capitalize the first letter of a String in Dart, you can use the following code.

```
//Example of capitalize first letter of String
void main() {
  String text = "hello world";
  print("Capitalized first letter of String:
${text[0].toUpperCase()}${text.substring(1)}");
}
```

Show Output

```
Capitalized first letter of String: Hello world
```

QUESTIONS FOR PRACTICE 1

Basic Dart Practice Questions

1. Write a program to print your name in Dart.

2. Write a program to print **Hello I am "John Doe"** and **Hello I'am "John Doe"** with single and double quotes.
3. Declare **constant type** of int set value 7.
4. Write a program in Dart that finds simple interest. `Formula= (p * t * r) / 100`
5. Write a program to print a square of a number using user input.
6. Write a program to print full name of a from first name and last name using user input.
7. Write a program to find quotient and remainder of two integers.
8. Write a program to swap two numbers.
9. Write a program in Dart to remove all whitespaces from String.
10. Write a Dart program to convert String to int.
11. Suppose, you often go to restaurant with friends and you have to split amount of bill. Write a program to calculate split amount of bill. `Formula= (total bill amount) / number of people`
12. Suppose, your distance to office from home is 25 km and you travel 40 km per hour. Write a program to calculate time taken to reach office in minutes. `Formula= (distance) / (speed)`

Dart Conditions and Loops

This section will help you to learn about the Conditions and Loops used in Dart. Here you will learn the following topics:

- Conditions in Dart,
- Loops in Dart,
- For Loop in Dart,
- For Each Loop in Dart,
- While Loop in Dart,
- Do While Loop in Dart,
- Switch Case in Dart,
- Break and Continue in Dart,
- Ternary Operator in Dart, and
- Exception Handling in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

CONDITIONS IN DART

Conditions In Dart

When you write a computer program, you need to be able to tell the computer what to do in different situations. With conditions, you can control the flow of the dart program. Suppose you need to execute a specific code when a particular situation is true. In that case, you can use conditions in Dart. E.g., a calculator app must perform subtraction if the user presses the subtract button and addition if the user taps the add button.

Types Of Condition

You can use following conditions to control the flow of your program.

- **If Condition**
- **If-Else Condition**
- **If-Else-If Condition**
- **Switch case**

If Condition

The easy and most common way of controlling the flow of a program is through the use of an *if statement*. If statement allow us to execute a code block when the given condition is true. Conditions evaluate boolean values.

Syntax

```
if(condition) {
    Statement 1;
    Statement 2;
        .
        .
    Statement n;
}
```

Example Of If Condition

It prints whether the person is a voter. If the person's age is greater and equal to 18, it will print, You are a voter.

```
void main()
{
    var age = 20;

    if(age >= 18){
      print("You are voter.");
    }
}
```

Show Output

```
You are voter.
```

If-Else Condition

If the result of the condition is true, then the body of the if-condition is executed. Otherwise, the body of the else-condition is executed.

Syntax

```
if(condition){
statements;
}else{
statements;
}
```

Example Of If-Else Condition

Dart program prints whether the person is a voter or not based on age.

```
void main(){
        int age = 12;
        if(age >= 18){
            print("You are voter.");
        }else{
            print("You are not voter.");
        }
}
```

Show Output

```
You are not voter.
```

Condition Based On Boolean Value

If the married status is false, it prints you are single; otherwise, it will print you are married.

```
void main(){
        bool isMarried = false;
        if(isMarried){
            print("You are married.");
        }else{
            print("You are single.");
        }
}
```

Show Output

```
You are single.
```

If-Else-If Condition

When you have multiple if conditions, then you can use if-else-if. You can learn more in the example below. When you have more than two conditions, you can use if, else if, else in dart.

Syntax

```
if(condition1){
statements1;
}else if(condition2){
statements2;
}else if(condition3){
statements3;
}
.
.
.
else(conditionN){
statementsN;
}
```

Example Of If-Else-If Condition

This program prints the month name based on the numeric value of that month. You will get a different result if you change the number of month.

```
void main() {
  int noOfMonth = 5;

  // Check the no of month
  if (noOfMonth == 1) {
    print("The month is jan");
  } else if (noOfMonth == 2) {
    print("The month is feb");
  } else if (noOfMonth == 3) {
    print("The month is march");
  } else if (noOfMonth == 4) {
    print("The month is april");
  } else if (noOfMonth == 5) {
    print("The month is may");
  } else if (noOfMonth == 6) {
    print("The month is june");
  } else if (noOfMonth == 7) {
    print("The month is july");
  } else if (noOfMonth == 8) {
    print("The month is aug");
  } else if (noOfMonth == 9) {
    print("The month is sep");
  } else if (noOfMonth == 10) {
    print("The month is oct");
  } else if (noOfMonth == 11) {
    print("The month is nov");
  } else if (noOfMonth == 12) {
    print("The month is dec");
  } else {
    print("Invalid option given.");
  }
}
```

Show Output

```
The month is may
```

Find Greatest Number Among 3 Numbers

Dart program, which finds the greatest number among three numbers.

```
void main(){
        int num1 = 1200;
        int num2 = 1000;
        int num3 = 150;

        if(num1 > num2  && num1 > num3){
            print("Num 1 is greater: i.e $num1");
        }
        if(num2 > num1 && num2 > num3){
           print("Num2 is greater: i.e $num2");
        }
        if(num3 > num1 && num3 > num2){
            print("Num3 is greater: i.e $num3");
        }
}
```

Show Output

```
Num 1 is greater: i.e 1200
```

ASSERT IN DART

Assert

While coding, it is hard to find errors in big projects, so dart provide a **assert** method to check for the error. It takes conditions as an argument. If the condition is true, nothing

happens. If a condition is false, it will raise an error.

Syntax

You can use assert without a custom message or with a custom message.

```
assert(condition);
// or
assert(condition, "Your custom message");
```

Example 1: How To Use Assert In Dart Program

This example shows how you can use assert without a custom message.

```
void main() {
   var age = 22;
   assert(age!=22);
}
```

Show Output

```
Uncaught Error: Assertion failed
```

Example 2: How To Use Assert In Dart Program

This example shows how you can use assert with a custom message.

```
void main() {
   var age = 22;
   assert(age!=22, "Age must be 22");
}
```

Show Output

```
Uncaught Error: Assertion failed: "Age must be 22"
```

How to Run File In Assert Mode

Use this command to run the dart file, which enables assert. If you don't use this, you will not be able to see the issue. You can use this command below if you are running a dart file from the computer.

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

Info

Note: The **assert(condition)** method only runs in development mode. It will throw an exception only when the condition is false. If the condition is true, nothing happens. Production code ignores it.

SWITCH CASE IN DART

Switch Case In Dart

In this tutorial, you will learn to use **dart switch case** to control your program's flow. A Switch case is used to execute the code block based on the condition.

```
switch(expression) {
    case value1:
        // statements
        break;
    case value2:
        // statements
        break;
    case value3:
        // statements
        break;
    default:
       // default statements
}
```

How does switch-case statement work in dart

- The **expression** is evaluated once and compared with each case value.
- If **expression** matches with case value1, the statements of case value1 are executed. Similarly, case value 2 will be executed if the expression matches case value2. If the expression matches the case value3, the statements of case value3 are executed.
- The **break** keywords tell dart to exit the switch statement because the statements in the case block are finished.
- If there is no match, **default statements** are executed.

Note: You can use a Switch case as an alternative to the **if-else-if** condition.

Replace If Else If With Switch In Dart

Here you can see the same program using **if else if** and **switch** in dart.

Example: Using If Else If

This example prints the day name based on the numeric day of the week using a if else if.

```
void main(){
   var dayOfWeek = 5;
if (dayOfWeek == 1) {
        print("Day is Sunday.");
  }
else if (dayOfWeek == 2) {
       print("Day is Monday.");
     }
else if (dayOfWeek == 3) {
      print("Day is Tuesday.");
     }
else if (dayOfWeek == 4) {
        print("Day is Wednesday.");
     }
else if (dayOfWeek == 5) {
        print("Day is Thursday.");
   }
else if (dayOfWeek == 6) {
        print("Day is Friday.");
    }
else if (dayOfWeek == 7) {
        print("Day is Saturday.");
}else{
        print("Invalid Weekday.");
     }
}
```

Show Output

```
Day is Thursday.
```

Example Of Switch Statement

This example prints the day name based on the numeric day of the week using a switch case.

```
void main() {
  var dayOfWeek = 5;
  switch (dayOfWeek) {
    case 1:
        print("Day is Sunday.");
        break;
    case 2:
        print("Day is Monday.");
      break;
    case 3:
      print("Day is Tuesday.");
      break;
    case 4:
        print("Day is Wednesday.");
      break;
    case 5:
        print("Day is Thursday.");
      break;
    case 6:
        print("Day is Friday.");
      break;
    case 7:
        print("Day is Saturday.");
      break;
    default:
        print("Invalid Weekday.");
      break;
  }
}
```

Show Output

```
Day is Thursday.
```

Note: The syntax of switch statements is cleaner and much easier to read and write.

Switch Case On Strings

You can also use a switch case with strings. This program prints information based on weather value.

```
void main() {
 const weather = "cloudy";

  switch (weather) {
    case "sunny":
        print("Its a sunny day. Put sunscreen.");
        break;
    case "snowy":
        print("Get your skis.");
      break;
    case "cloudy":
    case "rainy":
      print("Please bring umbrella.");
      break;
    default:
        print("Sorry I am not familiar with such weather.");
      break;
  }
}
```

Show Output

```
Please bring umbrella.
```

Switch Case On Enum

An **enum** or enumeration is used for defining value according to you. You can define your own type with a finite number of options. Here is the syntax for defining enum.

Syntax

```
enum enum_name {
  constant_value1,
  constant_value2,
  constant_value3
  }
```

Example of Switch Using Enum In Dart

Enum plays well with switch statements. Let’s see an example using enum.

```
// define enum outside main function
enum Weather{ sunny, snowy, cloudy, rainy}
// main method
void main() {
```

```
const weather = Weather.cloudy;

  switch (weather) {
    case Weather.sunny:
        print("Its a sunny day. Put sunscreen.");
        break;
    case Weather.snowy:
        print("Get your skis.");
      break;
    case Weather.rainy:
    case Weather.cloudy:
      print("Please bring umbrella.");
      break;
    default:
        print("Sorry I am not familiar with such weather.");
      break;
  }
}
```

Show Output

```
Please bring umbrella.
```

TERNARY OPERATOR IN DART

Ternary Operator

The ternary operator is like if-else statement. This is a one-liner replacement for the if-else statement. It is used to write a conditional expression, where based on the result of

a boolean condition, one of the two values is selected.

Syntax

```
condition ? exprIfTrue : exprIfFalse
```

Note: The ternary operator takes a condition and returns one of two values, depending upon the condition’s boolean value, i.e., true or false.

Ternary Operator Vs If Else

We already learned if-else in dart. Let us see the same example using the if-else and ternary operator.

Example Using If Else

This program finds greatest number between two numbers using if else.

```
void main() {
  int num1 = 10;
  int num2 = 15;
  int max = 0;
  if(num1> num2){
    max = num1;
  }else {
    max = num2;
  }
  print("The greatest number is $max");
}
```

Show Output

```
The greatest number is 15
```

Example 1: Using Ternary Operator

This program finds greatest number between two numbers using ternary operator.

```
void main() {
  int num1 = 10;
  int num2 = 15;
  int max = (num1 > num2) ? num1 : num2;
  print("The greatest number is $max");
}
```

Show Output

```
The greatest number is 15
```

Note: Ternary operator makes if-else code much shorter and readable. If you have problems with ternary, you can always use if-else.

Example 2: Ternary Operator Dart

If the selection value is 2 then it will set output as Apple otherwise, Banana.

```
void main() {
  var selection = 2;
  var output = (selection == 2) ? 'Apple' : 'Banana';
  print(output);
}
```

Show Output

```
Apple
```

Example 3 Ternary Operator Dart

This is a dart program to print whether the person is a voter or not using a ternary operator.

```
void main() {
  var age = 18;
  var check = (age >= 18) ? 'You ara a voter.' : 'You are not a voter.';
  print(check);
}
```

Show Output

```
You ara a voter.
```

Challenge

Create an int variable age and initialize it with your age. Write **ternary statement** to print “Teenager” if age is between 13 and 19 and “Not Teenager” if age is not between 13 and 19.

LOOPS IN DART

Dart Loops

In Programming, loops are used to repeat a block of code until certain conditions are not completed. For, e.g., if you want to print your name 100 times, then rather than typing print(“your name”); 100 times, you can use a loop.

There are different types of loop in Dart. They are:

- **For Loop**
- **For Each Loop**
- **While Loop**
- **Do While Loop**

Note: The primary purpose of all loops is to repeat a block of code.

Print Your Name 10 Times Without Using Loop

Let’s first print the name 10 times without using a loop.

```
void main() {
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
    print("John Doe");
}
```

Show Output

```
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
```

Print Your Name 10 Times Using Loop

We will learn for loop in the next section, paste the code and see the output. It will print your name 10 times.

```
void main() {
  for (int i = 0; i < 10; i++) {
    print("John Doe");
  }
}
```

Show Output

```
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
```

What if you want to print your name 1000 times? Without using a loop, it will be difficult for you.

Note: Loops are helpful because they reduce errors, save time, and make code more readable.

What After This?

Move to a new section to understand each loop clearly.

FOR LOOP IN DART

For Loop

This is the most common type of loop. You can use **for loop** to run a code block multiple times according to the condition. The syntax of for loop is:

```
for(initialization; condition; increment/decrement){
            statements;
}
```

- Initialization is executed (one time) before the execution of the code block.
- Condition defines the condition for executing the code block.
- Increment/Decrement is executed (every time) after the code block has been executed.

Example 1: To Print 1 To 10 Using For Loop

This example prints 1 to 10 using for loop. Here **int i = 1;** is initialization, **i<=10** is condition and **i++** is increment/decrement.

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

Show Output

```
1
2
3
4
5
6
7
8
9
10
```

Example 2: To Print 10 To 1 Using For Loop

This example prints 10 to 1 using for loop. Here **int i = 10;** is initialization, **i>=1** is condition and `i--` is increment/decrement.

```
void main() {
  for (int i = 10; i >= 1; i--) {
    print(i);
  }
}
```

Show Output

```
10
9
8
7
6
5
4
3
2
1
```

Example 3: Print Name 10 Times Using For Loop

This example prints the name 10 times using for loop. Based on the condition, the body of the loop executes 10 times.

```
void main() {
  for (int i = 0; i < 10; i++) {
    print("John Doe");
  }
}
```

Show Output

```
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
John Doe
```

Example 4: Display Sum of n Natural Numbers Using For Loop

Here, the value of the **total** is **0** initially. Then, the for loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the **total**, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

```
void main(){

  int total = 0;
  int n = 100; // change as per required

  for(int i=1; i<=n; i++){
    total = total + i;
  }

  print("Total is $total");

}
```

Show Output

```
Total is 5050
```

Example 5: Display Even Numbers Between 50 to 100 Using For Loop

This program will print even numbers between 50 to 100 using for loop.

```
void main(){
  for(int i=50; i<=100; i++){
    if(i%2 == 0){
      print(i);
    }
  }
}
```

Show Output

```
50
52
54
56
58
60
62
64
66
68
70
72
74
76
78
80
82
84
86
88
90
92
94
96
98
100
```

Infinite Loop In Dart

If the condition never becomes false in looping, it is called an infinite loop. It uses more resources on your computer. The task is done repeatedly until the memory runs out.

This program prints 1 to infinite because the condition is **i>=1**, which is always true with i++.

```
void main() {
  for (int i = 1; i >= 1; i++) {
    print(i);
  }
}
```

Note: Infinite loops take your computer resources continuously, use more power, and slow your computer. So always check your loop before use.

FOR EACH LOOP IN DART

For Each Loop

The **for each** loop iterates over all list elements or variables. It is useful when you want to loop through **list/collection**. The syntax of for-each loop is:

```
collection.forEach(void f(value));
```

Example 1: Print Each Item Of List Using Foreach

This will print each name of football players.

```
void main(){
  List<String> footballplayers=['Ronaldo','Messi','Neymar','Hazard'];
  footballplayers.forEach( (names)=>print(names));
}
```

Show Output

```
Ronaldo
Messi
Neymar
Hazard
```

Example 2: Print Each Total and Average Of Lists

This program will print the total sum of all numbers and also the average value from the total.

```
void main(){
  List<int> numbers = [1,2,3,4,5];
  int total = 0;

   numbers.forEach( (num)=>total= total+ num);

  print("Total is $total.");

  double avg = total / (numbers.length);

  print("Average is $avg.");

}
```

Show Output

```
Total is 15.
Average is 3.
```

For In Loop In Dart

There is also another for loop, i.e., **for in loop**. It also makes looping over the list very easily.

```
void main(){
    List<String> footballplayers=['Ronaldo','Messi','Neymar','Hazard'];

  for(String player in footballplayers){
    print(player);
  }
}
```

Show Output

```
Ronaldo
Messi
Neymar
Hazard
```

How to Find Index Value Of List

In dart, asMap method converts the list to a map where the keys are the index and values are the element at the index.

```
void main(){

List<String> footballplayers=['Ronaldo','Messi','Neymar','Hazard'];

footballplayers.asMap().forEach((index, value) => print("$value index is $index"));

}
```

Show Output

```
Ronaldo index is 0
Messi index is 1
Neymar index is 2
Hazard index is 3
```

Example 3: Print Unicode Value of Each Character of String

This will split the name into Unicode values and then find characters from the Unicode value.

```
void main(){

String name = "John";

for(var codePoint in name.runes){
  print("Unicode of ${String.fromCharCode(codePoint)} is $codePoint.");
}
}
```

Show Output

```
Unicode of J is 74.
Unicode of o is 111.
Unicode of h is 104.
Unicode of n is 110.
```

WHILE LOOP IN DART

While Loop

In **while loop**, the loop's body will run until and unless the condition is true. You must write conditions first before statements. This loop checks conditions on every iteration. If the condition is true, the code inside {} is executed, if the condition is false, then the loop stops.

Syntax

```
while(condition){
       //statement(s);
      // Increment (++) or Decrement (--) Operation;
}
```

- A while loop evaluates the condition inside the parenthesis ().
- If the condition is true, the code inside {} is executed.
- The condition is re-checked until the condition is false.
- When the condition is false, the loop stops.

Example 1: To Print 1 To 10 Using While Loop

This program prints 1 to 10 using while loop.

```
void main() {
  int i = 1;
  while (i <= 10) {
    print(i);
    i++;
  }
}
```

Show Output

```
1
2
3
4
5
6
7
8
9
10
```

Note: Do not forget to increase the variable used in the condition. Otherwise, the loop will never end and becomes an infinite loop.

Example 2: To Print 10 To 1 Using While Loop

This program prints 10 to 1 using while loop.

```
void main() {
  int i = 10;
  while (i >= 1) {
    print(i);
    i--;
  }
}
```

Show Output

```
10
9
8
7
6
5
4
```

```
3
2
1
```

Example 3: Display Sum of n Natural Numbers Using While Loop

Here, the value of the total is 0 initially. Then, the while loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the total, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

```
void main(){

  int total = 0;
  int n = 100; // change as per required
  int i =1;

  while(i<=n){
    total = total + i;
    i++;
  }

  print("Total is $total");

}
```

Show Output

```
Total is 5050
```

Example 4: Display Even Numbers Between 50 to 100 Using While Loop

This program will print even numbers between 50 to 100 using while loop.

```
void main(){
  int i = 50;
  while(i<=100){
  if(i%2 == 0){
      print(i);
    }
    i++;
  }
}
```

Show Output

```
50
52
54
56
58
60
```

```
62
64
66
68
70
72
74
76
78
80
82
84
86
88
90
92
94
96
98
100
```

DO WHILE LOOP IN DART

Do While Loop

Do while loop is used to run a block of code multiple times. The loop's body will be executed first, and then the condition is tested. The syntax of do while loop is:

```
do{
    statement1;
    statement2;
    .
    .
    .
    statementN;
}while(condition);
```

- First, it runs statements, and finally, the condition is checked.
- If the condition is true, the code inside {} is executed.
- The condition is re-checked until the condition is false.
- When the condition is false, the loop stops.

Note: In a do-while loop, the statements will be executed at least once time, even if the condition is false. It is because the statement is executed before checking the condition.

Example 1: To Print 1 To 10 Using Do While Loop

```
void main() {
  int i = 1;
  do {
    print(i);
    i++;
  } while (i <= 10);
}
```

Show Output

```
1
2
3
4
5
6
7
8
9
10
```

Example 2: To Print 10 To 1 Using Do While Loop

```
void main() {
  int i = 10;
  do {
    print(i);
    i--;
  } while (i >= 1);
}
```

Show Output

```
10
9
8
7
6
5
4
3
2
1
```

Example 3: Display Sum of n Natural Numbers Using Do While Loop

Here, the value of the **total** is 0 initially. Then, the do-while loop is iterated from **i = 1 to 100**. In each iteration, **i** is added to the total, and the value of **i** is increased by 1. Result is **1+2+3+….+99+100**.

```
void main(){

  int total = 0;
  int n = 100; // change as per required
  int i =1;

  do{
  total = total + i;
    i++;
  }while(i<=n);

  print("Total is $total");

}
```

Show Output

```
Total is 5050
```

When The Condition Is False

Let’s make one condition false and see the demo below. **Hello** got printed if the condition is false.

```
void main(){

  int number = 0;

  do{
  print("Hello");
  number--;
  }while(number >1);

}
```

Show Output

```
Hello
```

BREAK AND CONTINUE IN DART

Dart Break and Continue

In this tutorial, you will learn about the **break and continue** in dart. While working on loops, we need to skip some elements or terminate the loop immediately without checking the condition. In such a situation, you can use the break and continue statement.

Break Statement

Sometimes you will need to break out of the loop immediately without checking the condition. You can do this using break statement.

The break statement is used to exit a loop. It stops the loop immediately, and the program’s control moves outside the loop. Here is syntax of break:

```
break;
```

Example 1: Break In Dart For Loop

Here, the loop condition is true until the value of i is less than or equal to 10. However, the break says to go outside the loop when the value of i becomes 5.

```
void main() {
  for (int i = 1; i <= 10; i++) {
    if (i == 5) {
      break;
    }
    print(i);
  }
}
```

Show Output

```
1
2
3
4
```

Example 2: Break In Dart Negative For Loop

Here, the loop condition is true until the value of i is more than or equal to 1. However, the break says to go outside the loop when the value of i becomes 7.

```
void main() {
  for (int i = 10; i >= 1; i--) {
    if (i == 7) {
      break;
    }
    print(i);
  }
}
```

Show Output

```
10
9
8
```

Example 3: Break In Dart While Loop

Here, this while loop condition is true until the value of i is less than or equal to 10. However, the break says to go outside the loop when the value of i becomes 5.

```
void main() {
 int i =1;
 while(i<=10){
  print(i);
   if (i == 5) {
      break;
    }
    i++;
 }
}
```

Show Output

```
1
2
3
4
5
```

Example 4: Break In Switch Case

As we already learn in dart switch case, it is important to add **break** keyword in switch statement. This example prints the month name based on the number of the month using a switch case.

```
void main() {
  var noOfMoneth = 5;
  switch (noOfMoneth) {
    case 1:
      print("Selected month is January.");
      break;
    case 2:
      print("Selected month is February.");
      break;
    case 3:
      print("Selected month is march.");
      break;
    case 4:
      print("Selected month is April.");
      break;
    case 5:
      print("Selected month is May.");
      break;
    case 6:
      print("Selected month is June.");
      break;
    case 7:
      print("Selected month is July.");
      break;
    case 8:
      print("Selected month is August.");
      break;
    case 9:
      print("Selected month is September.");
      break;
    case 10:
      print("Selected month is October.");
      break;
    case 11:
      print("Selected month is November.");
      break;
    case 12:
      print("Selected month is December.");
      break;
    default:
      print("Invalid month.");
      break;
  }
}
```

Show Output

```
Selected month is May.
```

Continue Statement

Sometimes you will need to skip an iteration for a specific condition. You can do this utilizing continue statement.

The continue statement skips the current iteration of a loop. It will bypass the statement of the loop. It does not terminate the loop but rather continues with the next iteration. Here is the syntax of continue statement:

```
continue;
```

Example 1: Continue In Dart

Here, the loop condition is true until the value of i is less than or equal to 10. However, the continue says to go to the next iteration of the loop when the value of i becomes 5.

```
void main() {
  for (int i = 1; i <= 10; i++) {
    if (i == 5) {
      continue;
    }
    print(i);
  }
}
```

Show Output

```
1
```

```
2
3
4
6
7
8
9
10
```

Example 2: Continue In For Loop Dart

Here, the loop condition is true until the value of i is more than or equal to 1. However, the continue says to go to the next iteration of the loop when the value of i becomes 4.

```
void main() {
  for (int i = 10; i >= 1; i--) {
    if (i == 4) {
      continue;
    }
    print(i);
  }
}
```

Show Output

```
10
9
8
7
6
5
3
2
1
```

Example 3: Continue In Dart While Loop

Here, this while loop condition is true until the value of i is less than or equal to 10. However, the continue says to go to the next iteration of the loop when the value of i becomes 5.

```
void main() {
  int i = 1;
  while (i <= 10) {
    if (i == 5) {
      i++;
      continue;
    }
    print(i);
    i++;
  }
}
```
Show Output

```
1
2
3
4
6
7
8
9
10
```

EXCEPTION HANDLING IN DART

Exception In Dart

An exception is an error that occurs at runtime during program execution. When the exception occurs, the flow of the program is interrupted, and the program terminates abnormally. There is a high chance of crashing or terminating the program when an exception occurs. Therefore, to save your program from crashing, you need to catch the exception.

Note: If you are attempting a task that might result in an error, it's a good habit to use the try-catch statement.

Syntax

```
try {
// Your Code Here
  }
catch(ex){
// Exception here
}
```

Try & Catch In Dart

Try You can write the logical code that creates exceptions in the try block.

Catch When you are uncertain about what kind of exception a program produces, then a catch block is used. It is written with a try block to catch the general exception.

Example 1: Try Catch In Dart

In this example, you will see how to handle the exception using the try-catch block.

```
void main() {
   int a = 18;
   int b = 0;
   int res;

   try {
      res = a ~/ b;
      print("Result is $res");
   }
    // It returns the built-in exception related to the occurring
exception
   catch(ex) {
      print(ex);
    }
}
```

Show Output

```
IntegerDivisionByZeroException
```

Finally In Dart Try Catch

The **finally** block is always executed whether the exceptions occur or not. It is optional to include the final block, but if it is included, it should be after the try and catch block is over.

On block is used when you know what types of exceptions are produced by the program.

Syntax

```
try {
.....
}
on Exception1 {
....
}
catch Exception2 {
....
}
finally {
// code that should always execute whether an exception or not.
}
```

Example 2: Finally In Dart Try Catch

In this example, you will see how to handle the exception using the try-catch block with the finally block.

```
void main() {
  int a = 12;
  int b = 0;
  int res;
  try {
    res = a ~/ b;
  } on UnsupportedError {
    print('Cannot divide by zero');
  } catch (ex) {
    print(ex);
  } finally {
    print('Finally block always executed');
  }
}
```

Show Output

```
Cannot divide by zero
Finally block always executed
```

Throwing An Exception

The throw keyword is used to raise an exception explicitly. A raised exception should be handled to prevent the program from exiting unexpectedly.

Syntax

```
throw new Exception_name()
```

Example 3: Throwing An Exception

In this example, you will see how to throw an exception using the throw keyword.

```
void main() {
  try {
    check_account(-10);
  } catch (e) {
    print('The account cannot be negative');
  }
}

void check_account(int amount) {
  if (amount < 0) {
    throw new FormatException(); // Raising explanation externally
  }
}
```

Show Output

```
The account cannot be negative
```

Why Is Exception Handling Needed?

Exceptions provide the means to separate the details of what to do when something out of the ordinary happens from the main logic of a program. Therefore, exceptions must be handled to prevent the application from unexpected termination. Here are some reasons why exception handling is necessary:

- To avoid abnormal termination of the program.
- To avoid an exception caused by logical error.
- To avoid the program from falling apart when an exception occurs.
- To reduce the vulnerability of the program.
- To maintain a good user experience.
- To try providing aid and some debugging in case of an exception.

How To Create Custom Exception In Dart

As you go advance, you need to create your exception; Dart enables you to create your exception.

Syntax

```
class YourExceptionClass implements Exception{
  // constructors, variables & methods
}
```

Example 4: How to Create & Handle Exception

This program throws an exception when a student’s mark is negative. You will understand **implements** in the object-oriented programming section.

```
class MarkException implements Exception {
  String errorMessage() {
    return 'Marks cannot be negative value.';
  }
}

void main() {
  try {
    checkMarks(-20);
  } catch (ex) {
    print(ex.toString());
  }
}

void checkMarks(int marks) {
  if (marks < 0) throw MarkException().errorMessage();
}
```

Show Output

```
Marks cannot be negative value.
```

Example 5: How to Create & Handle Exception

This program throws an exception when you find the square root of a negative number.

```dart
import 'dart:math';

// custom exception class
class NegativeSquareRootException implements Exception {
  @override
  String toString() {
    return 'Sqauare root of negative number is not allowed here.';
  }
}

// get square root of a positive number
num squareRoot(int i) {
  if (i < 0) {
    // throw `NegativeSquareRootException` exception
    throw NegativeSquareRootException();
  } else {
    return sqrt(i);
  }
}

void main() {
  try {
    var result = squareRoot(-4);

    print("result: $result");
  } on NegativeSquareRootException catch (e) {
    print("Oops, Negative Number: $e");
  } catch (e) {
    print(e);
  } finally {
    print('Job Completed!');
  }
}
```

Show Output

```
Oops, Negative Number: Sqauare root of negative number is not allowed
here.
Job Completed!
```

QUESTION FOR PRACTICE 2

Question For Practice 2

1. Write a dart program to check if the number is odd or even.
2. Write a dart program to check whether a character is a vowel or consonant.
3. Write a dart program to check whether a number is positive, negative, or zero.
4. Write a dart program to print your name 100 times.

5. Write a dart program to calculate the sum of natural numbers.
6. Write a dart program to generate multiplication tables of 5.
7. Write a dart program to generate multiplication tables of 1-9.
8. Write a dart program to create a simple calculator that performs addition, subtraction, multiplication, and division.
9. Write a dart program to print 1 to 100 but not 41.

Dart Functions

This section will help you to learn about the function in Dart. Here you will learn the following topics:

- Function in Dart,
- Types of Function in Dart,
- Function Parameter in Dart,
- Anonymous Function in Dart,
- Arrow Function in Dart,
- Scope in Dart, and
- Math in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge. Improve your coding skills with the help of these questions.

Accelerate your Workflow

Save any snippet in this tutorial to your personal micro-repository with Pieces for Developers to speed up your workflow. Pieces is a centralized productivity suite that leverages AI to help developers save snippets, extract code from screenshots, auto-enrich code, and much more.

FUNCTIONS IN DART

Function In Dart

In this tutorial, you will learn about functions in dart. **Functions** are the block of code that performs a specific task. They are created when some statements are repeatedly occurring in the program. The function helps reusability of the code in the program.

Note: The main objective of the function is **DRY(Don't Repeat Yourself)**.

Function Advantages

- Avoid Code Repetition
- Easy to divide the complex program into smaller parts
- Helps to write a clean code

Syntax

```
returntype functionName(parameter1,parameter2, ...){
  // function body
}
```

Return type:

It tells you the function output type. It can be void, String, int, double, etc. If the function doesn't return anything, you can use void as the return type.

Function Name:

You can name functions by almost any name. Always follow a lowerCamelCase naming convention like void printName().

Parameters:

Parameters are the input to the function, which you can write inside the bracket (). Always follow a lowerCamelCase naming convention for your function parameter.

Example 1: Function That Prints Name

This is a simple program that prints name using function. The name of function is **printName()**.

```
// writing function outside main function.
void printName(){
  print("My name is Raj Sharma. I am from function.");
}
// this is our main function.
void main(){
  printName();
}
```

Show Output

```
My name is Raj Sharma. I am from function.
```

Example 2: Function To Find Sum of Two Numbers

This function finds the sum of two numbers. Here, the function accepts two parameters. i.e., **num1 and num2**, and the return type is void.

```
void add(int num1, int num2){
  int sum = num1 + num2;
   print("The sum is $sum");
}

void main(){
  add(10, 20);
}
```

Show Output

```
The sum is 30
```

Example 3: Function That Find Simple Interest

This function finds simple interest from principal, time and rate and display result.

```
// function that calculate interest
void calculateInterest(double principal, double rate, double time) {
  double interest = principal * rate * time / 100;
  print("Simple interest is $interest");
}

void main() {
  double principal = 5000;
  double time = 3;
  double rate = 3;
  calculateInterest(principal, rate, time);
}
```

Show Output

```
Simple interest is 450.0
```

Challenge

Create a function that finds a cube of numbers.

Key Points

- In dart function are also objects.
- You should follow the **lowerCamelCase** naming convention while naming function.
- You should follow the **lowerCamelCase** naming convention while naming function parameters.

About lowerCamelCase

Name should start with lower-case, and every second word's first letter will be upper-case like num1, fullName, isMarried, etc. Technically, this naming convention is called lowerCamelCase.

Function Parameters Vs Arguments

Many programmers are often confused about parameters and arguments. Let's have a look at this example.

```
// Here num1 and num2 are parameters
void add(int num1, int num2){
  int sum;
  sum = num1 + num2;

  print("The sum is $sum");
}

void main(){
// Here 10 and 20 are arguments
  add(10, 20);
}
```

Show Output

```
The sum is 30
```

- Here in **add(int num1, int num2)**, num1 and num2 are parameters and in **add(10, 20)**, 10 and 20 are arguments.
- Parameter is the name and data type you define as an input for your function.
- Argument is the actual value that you passed in.

Note: In dart, if you don't write the return type of function. It will automatically understand.

TYPES OF FUNCTIONS IN DART

Types Of Function

Functions are the block of code that performs a specific task. Here are different types of functions:

- No Parameter And No Return Type
- Parameter And No Return Type
- No Parameter And Return Type
- Parameter And Return Type

Function With No Parameter And No Return Type

In this function, you do not pass any parameter and expect no return type. Here is an example of it:

Example 1: No Parameter & No Return Type

Here **printName()** is a function which prints name on screen.

```
void main() {
  printName();
}
```

```
void printName() {
  print("My name is John Doe.");
}
```

Show Output

```
My name is John Doe.
```

In this program, **printName()** is the function which has keyword **void**. It means it has **no return type**, and the empty pair of parentheses implies that there is **no parameter** that is passed to the function.

Example 2: No Parameter & No Return Type

Here **printPrimeMinisterName()** is a function which prints prime minister name on screen.

```
void main() {
  print("Function With No Parameter and No Return Type");
  printPrimeMinisterName();
}

void printPrimeMinisterName() {
  print("John Doe.");
}
```

Show Output

```
Function With No Parameter and No Return Type
John Doe.
```

Function With Parameter And No Return Type

In this function, you do pass the parameter and expect no return type. Here is an example of it:

Example 1: Parameter & No Return Type

Here **printName(String name)** is a function which welcome person.

```
void main() {
  printName("John");
}

void printName(String name) {
  print("Welcome, ${name}.");
}
```

Show Output

```
Welcome, John.
```

In this program, **printName(String name)** is the function which has keyword **void**. It means it has **no return type**, and the pair of parentheses is not empty but this time that suggests it to accept an **parameter**.

Example 2: Parameter & No Return Type

Here **add(int a, int b)** is a function that finds and prints the sum of two numbers.

```
// This function add two numbers
void add(int a, int b) {
  int sum = a + b;
  print("The sum is $sum");
}

void main() {
  int num1 = 10;
  int num2 = 20;

  add(num1, num2);
}
```

Show Output

```
The sum is 30
```

Function With No Parameter And Return Type

In this function, you do not pass any parameter but expect return type. Here is an example of it:

Example 1: No Parameter & Return Type

Here **primeMinisterName()** is a function which returns prime minister name. In the entire program, anyone can use this function to find the name of the prime minister.

```
void main() {
// Function With No Parameter & Return Type
  String name = primeMinisterName();
  print("The Name from function is $name.");
}
String primeMinisterName() {
  return "John Doe";
}
```

Show Output

```
The name from function is John Doe
```

In this program, **primeMinisterName()** is the function which has **String** keyword before function name, means it **return** String value, and the empty pair of parentheses suggests that there is **no parameter** that is passed to the function.

Example 2: No Parameter & Return Type

Here **voterAge()** is a function which returns minimum voter age.

```
// Function With No Parameter & Return Type
void main() {
  int personAge = 17;

  if (personAge >= voterAge()) {
    print("You can vote.");
  } else {
    print("Sorry, you can't vote.");
  }
}

int voterAge() {
  return 18;
}
```

Show Output

```
Sorry, you can't vote.
```

Function With Parameter And Return Type

In this function, you do pass the parameter and also expect return type. Here is an example of it:

Example 1: Parameter & Return Type

Here **add(int a, int b)** is a function that returns its sum in integer. We can display results in our main function.

```
// this function add two numbers
int add(int a, int b) {
  int sum = a + b;
  return sum;
}

void main() {
  int num1 = 10;
  int num2 = 20;

  int total = add(num1, num2);
  print("The sum is $total.");
}
```
Show Output

```
The sum is 30.
```

In this program, **int add(int a, int b)** is the function with **int** as the return type, and the pair of parenthesis has two **parameters**, i.e., a and b.

Example 2: Parameter & Return Type

Here **calculateInterest(double principal, double rate, double time)** is a function that returns its simple interest in double. We can display results in our main function.

```
// function that calculate interest
double calculateInterest(double principal, double rate, double time) {
  double interest = principal * rate * time / 100;
  return interest;
}

void main() {
  double principal = 5000;
  double time = 3;
  double rate = 3;
  double result = calculateInterest(principal, rate, time);
  print("The simple interest is $result.");
}
```
Show Output

```
The simple interest is 450.0.
```

Note: void is used for no return type as it is a non value-returning function.

**Complete Example **

Here is the program, which includes all types of functions we studied earlier.

```
// parameter and return type
int add(int a, int b) {
  var total;
  total = a + b;
  return total;
}

// parameter and no return type
void mul(int a, int b) {
  var total;
  total = a * b;
  print("Multiplication is : $total");
}

// no parameter and return type
String greet() {
  String greet = "Welcome";
  return greet;
}

// no parameter and no return type
void greetings() {
  print("Hello World!!!");
}

void main() {
  var total = add(2, 3);
  print("Total sum: $total");
  mul(2, 3);
  var greeting = greet();
  print("Greeting: $greeting");
  greetings();
}
```

Show Output

```
Total sum: 5
Multiplication is : 6
Greeting: Welcome
Hello World!!!
```

FUNCTION PARAMETER IN DART

Parameter In Dart

The parameter is the process of passing values to the function. The values passed to the function must match the number of parameters defined. A function can have any number of parameters.

```
// here a and b are parameters
void add(int a, int b) {
}
```

Positional Parameter In Dart

In positional parameters, you must supply the arguments in the same order as you defined on parameters when you wrote the function. If you call the function with the parameter in the wrong order, you will get the wrong result.

Example 1: Use Of Positional Parameter

In the example below, the function **printInfo** takes two parameters. You must pass the person's name and gender in the same order. If you pass values in the wrong order, you will get the **wrong result**.

```
void printInfo(String name, String gender) {
  print("Hello $name your gender is $gender.");
}

void main() {
  // passing values in wrong order
  printInfo("Male", "John");

  // passing values in correct order
  printInfo("John", "Male");

}
```

Show Output

```
Hello Male your gender is John.
Hello John your gender is Male.
```

Example 2: Providing Default Value On Positional Parameter

In the example below, function **printInfo** takes two positional parameters and one optional parameter. The title parameter is optional here. If the user doesn't pass the title, it will automatically set the title value to **sir/ma'am**.

```
void printInfo(String name, String gender, [String title = "sir/ma'am"]) {
  print("Hello $title $name your gender is $gender.");
}

void main() {
  printInfo("John", "Male");
  printInfo("John", "Male", "Mr.");
  printInfo("Kavya", "Female", "Ms.");
}
```

Show Output

```
Hello sir/ma'am John your gender is Male.
Hello Mr. John your gender is Male.
Hello Ms. Kavya your gender is Female.
```

Example 3: Providing Default Value On Positional Parameter

In the example below, function **add** takes two positional parameters and one optional parameter. The **num3** parameter is **optional** here with default value **0**.

```
void add(int num1, int num2, [int num3=0]){
   int sum;
  sum = num1 + num2 + num3;

   print("The sum is $sum");
}

void main(){
  add(10, 20);
  add(10, 20, 30);
}
```

Show Output

```
The sum is 30
The sum is 60
```

Named Parameter In Dart

Dart allows you to use named parameters to clarify the parameter's meaning in function calls. **Curly braces {}** are used to specify named parameters.

Example 1: Use Of Named Parameter

In the example below, function **printInfo** takes two named parameters. You can pass value in any order. You will learn about **?** in **null safety** section.

```
void printInfo({String? name, String? gender}) {
  print("Hello $name your gender is $gender.");
}

void main() {
  // you can pass values in any order in named parameters.
  printInfo(gender: "Male", name: "John");
  printInfo(name: "Sita", gender: "Female");
  printInfo(name: "Reecha", gender: "Female");
  printInfo(name: "Reecha", gender: "Female");
  printInfo(name: "Harry", gender: "Male");
  printInfo(gender: "Male", name: "Santa");
}
```

Show Output

```
Hello John your gender is Male.
Hello Sita your gender is Female.
Hello Reecha your gender is Female.
Hello Reecha your gender is Female.
Hello Harry your gender is Male.
Hello Santa your gender is Male.
```

Example 2: Use Of Required In Named Parameter

In the example below, function **printInfo** takes two named parameters. You can see a **required** keyword, which means you must pass the person's name and gender. If you don't pass it, it won't work.

```
void printInfo({required String name, required String gender}) {
  print("Hello $name your gender is $gender.");
}

void main() {
  // you can pass values in any order in named parameters.
  printInfo(gender: "Male", name: "John");
  printInfo(gender: "Female", name: "Suju");
}
```

Show Output

```
Hello John your gender is Male.
Hello Suju your gender is Female.
```

Note: You can pass the value in any order in the named parameter. **?** is used to remove null safety, which we will discuss in the coming chapter.

Optional Parameter In Dart

Dart allows you to use optional parameters to make the parameter optional in function calls. **Square braces []** are used to specify optional parameters.

Example: Use Of Optional Parameter

In the example below, function **printInfo** takes two **positional parameters** and one **optional parameter**. First, you must pass the person's name and gender. The title parameter is optional here. Writing **[String? title]** makes **title** optional.

```
void printInfo(String name, String gender, [String? title]) {
  print("Hello $title $name your gender is $gender.");
}

void main() {
  printInfo("John", "Male");
  printInfo("John", "Male", "Mr.");
  printInfo("Kavya", "Female", "Ms.");
}
```

Show Output

```
Hello  John your gender is Male.
Hello Mr. John your gender is Male.
Hello Ms. Kavya your gender is Female.
```

ANONYMOUS FUNCTION IN DART

Anonymous Function In Dart

This tutorial will teach you the anonymous function and how to use it. You already saw function like **main()**, **add()**, etc. These are the **named** functions, which means they have a certain name.

But not every function needs a name. If you remove the return type and the function name, the function is called **anonymous function**.

Syntax

Here is the syntax of the anonymous function.

```
(parameterList){
// statements
}
```

Example 1: Anonymous Function In Dart

In this example, you will learn to use an anonymous function to print all list items. This function invokes each fruit without having a function name.

```
void main() {
  const fruits = ["Apple", "Mango", "Banana", "Orange"];

  fruits.forEach((fruit) {
    print(fruit);
  });
}
```

Show Output

```
Apple
Mango
Banana
Orange
```

Example 2: Anonymous Function In Dart

In this example, you will learn to find the cube of a number using an anonymous function.

```
void main() {
// Anonymous function
  var cube = (int number) {
    return number * number * number;
  };

  print("The cube of 2 is ${cube(2)}");
  print("The cube of 3 is ${cube(3)}");
}
```

Show Output

```
The cube of 2 is 8
The cube of 3 is 27
```

ARROW FUNCTION IN DART

Arrow Function In Dart

Dart has a special syntax for the function body, which is only one line. The arrow function is represented by **=>** symbol. It is a shorthand syntax for any function that has only one expression.

Syntax

The syntax for the dart arrow function.

```
returnType functionName(parameters...) => expression;
```

Note: The arrow function is used to make your code short.**=> expr** syntax is a shorthand for **{ return expr; }**.

Example 1: Simple Interest Without Arrow Function

This program finds simple interest without using the arrow function.

```
// function that calculate interest
double calculateInterest(double principal, double rate, double time) {
  double interest = principal * rate * time / 100;
  return interest;
}

void main() {
  double principal = 5000;
  double time = 3;
  double rate = 3;

  double result = calculateInterest(principal, rate, time);
  print("The simple interest is $result.");
}
```

Show Output

```
Simple interest is 450.0
```

Example 2: Simple Interest With Arrow Function

This program finds simple interest using the arrow function.

```
// arrow function that calculate interest
double calculateInterest(double principal, double rate, double time) =>
    principal * rate * time / 100;

void main() {
  double principal = 5000;
  double time = 3;
  double rate = 3;

  double result = calculateInterest(principal, rate, time);
  print("The simple interest is $result.");
}
```

Show Output

```
Simple interest is 450.0
```

Example 3: Simple Calculation Using Arrow Function

This program finds the sum, difference, multiplication, and division of two numbers using the arrow function.

```
int add(int n1, int n2) => n1 + n2;
int sub(int n1, int n2) => n1 - n2;
int mul(int n1, int n2) => n1 * n2;
double div(int n1, int n2) => n1 / n2;

void main() {
  int num1 = 100;
  int num2 = 30;

  print("The sum is ${add(num1, num2)}");
  print("The diff is ${sub(num1, num2)}");
  print("The mul is ${mul(num1, num2)}");
  print("The div is ${div(num1, num2)}");
}
```

Show Output

```
The sum is 130
The diff is 70
The mul is 3000
The div is 3.3333333333333335
```

SCOPE IN DART

Scope In Dart

The scope is a concept that refers to where values can be accessed or referenced. Dart uses curly braces **{}** to determine the scope of variables. If you define a variable inside curly braces, you can't use it outside the curly braces.

Method Scope

If you created variables inside the method, you can use them inside the method block but not outside the method block.

Example 1: Method Scope

```
void main() {
  String text = "I am text inside main. Anyone can't access me.";
  print(text);
}
```

Show Output

```
I am text inside main. Anyone can't access me.
```

In this program, **text** is a String type where you can access and print method only inside the main function but not outside the main function.

Global Scope

You can define a variable in the global scope to use the variable anywhere in your program.

Example 1: Global Scope

```
String global = "I am Global. Anyone can access me.";
void main() {
  print(global);
}
```

Show Output

```
I am Global. Anyone can access me.
```

In this program, the variable named **global** is a top-level variable; you can access it anywhere in the program.

Info

Note: Define your variable as much as close **Local** as you can. It makes your code clean and prevents you from using or changing them where you shouldn't.

Lexical Scope

Dart is lexically scoped language, which means you can find the scope of variables with the help of **braces {}**.

MATH IN DART

Math In Dart

Math helps you to perform mathematical calculations efficiently. With dart math, you can **generate random number**, **find square root**, **find power of number**, or **round specific numbers**. To use math in dart, you must `import 'dart:math';`.

How To Generate Random Numbers In Dart

This example shows how to generate random numbers from **0 - 9** and also **1 to 10**. After watching this example, you can generate a random number between your choices.

```
import 'dart:math';
void main()
{
Random random = new Random();
int randomNumber = random.nextInt(10); // from 0 to 9 included
print("Generated Random Number Between 0 to 9: $randomNumber");

int randomNumber2 = random.nextInt(10)+1; // from 1 to 10 included
print("Generated Random Number Between 1 to 10: $randomNumber2");
}
```

- In this program, **random.nextInt(10)** function is used to generate a random number between **0 and 9** in which the value is stored in a variable **randomNumber**.
- The **random.nextInt(10)+1** function is used to generate random number between **1 to 10** in which the value is stored in a variable **randomNumber2**.

Generate Random Number Between Any Number

Use this formula to generate a random number between any numbers in the dart.

```
 min + Random().nextInt((max + 1) - min);
```

Example: Random Number In Dart Between 10 - 20

This program generates random numbers between 10 to 20.

```
import 'dart:math';
void main()
```

```
{

int min = 10;
int max = 20;

int randomnum = min + Random().nextInt((max + 1) - min);

print("Generated Random number between $min and $max is: $randomnum");
}
```

Show Output

```
Generated Random number between 10 and 20 is 19
```

Random Boolean And Double Value

Here you will learn how to generate random boolean and double values in dart.

```
  Random().nextBool(); // return true or false
  Random().nextDouble(); // return 0.0 to 1.0
```

Example 1: Generate Random Boolean And Double Values

This example below generate random and boolean value.

```
import 'dart:math';
void main()
{
double randomDouble = Random().nextDouble();
bool randomBool = Random().nextBool();

print("Generated Random double value is: $randomDouble");
print("Generated Random bool value is: $randomBool");
}
```

Example 2: Generate a List Of Random Numbers In Dart

This example will generate a list of 10 random numbers between 1 to 100.

```
import 'dart:math';
void main()
{
List<int> randomList = List.generate(10, (_) => Random().nextInt(100)+1);
print(randomList);
}
```

Useful Math Function In Dart

You can use some useful math functions to perform your daily task with dart programming.

Function Name	Output	Description
pow(10,2)	100	10 to the power 2 is 10*10
max(10,2)	10	Maximum number is 10
min(10,2)	2	Minimum number is 2
sqrt(25)	5	Square root of 25 is 5

Example: Math In Dart

This example below finds the power of a number, a minimum and maximum value between two numbers, and the square root of a number.

```
import 'dart:math';
void main()
{
  int num1 = 10;
  int num2 = 2;

  num powernum = pow(num1,num2);
  num maxnum = max(num1,num2);
  num minnum = min(num1,num2);
  num squareroot = sqrt(25); // Square root of 25

  print("Power is $powernum");
  print("Maximum is $maxnum");
  print("Minimum is $minnum");
  print("Square root is $squareroot");

}
```

Show Output

```
Power is 100
Maximum is 10
Minimum is 2
Square root is 5.0
```

- In this program, **pow(num1, num2)** is a function where num1 is a digit and num2 is a power.
- **max(num1,num2)** is a function which give the maximum number between num1 and num2.
- **min(num1,num2)** is a function which give the mininum number between num1 and num2.
- **sqrt(25)** is a function that gives the square root of 25.

QUESTIONS FOR PRACTICE 3

Question For Practice 3

1. Write a program in Dart to print your own name using function.
2. Write a program in Dart to print even numbers between intervals using function.
3. Create a function called greet that takes a name as an argument and prints a greeting message. For example, greet(“John”) should print “Hello, John”.
4. Write a program in Dart that generates random password.
5. Write a program in Dart that find the area of a circle using function. Formula: pi * r * r
6. Write a program in Dart to reverse a String using function.
7. Write a program in Dart to calculate power of a certain number. For e.g 5^3=125
8. Write a function in Dart named add that takes two numbers as arguments and returns their sum.
9. Write a function in Dart called maxNumber that takes three numbers as arguments and returns the largest number.
10. Write a function in Dart called isEven that takes a number as an argument and returns True if the number is even, and False otherwise.
11. Write a function in Dart called createUser with parameters name, age, and isActive, where isActive has a default value of true.
12. Write a function in Dart called calculateArea that calculates the area of a rectangle. It should take length and width as arguments, with a default value of 1 for both. Formula: length * width.

Dart Collection

This section will help you to learn about the collection in Dart. Here you will learn the following topics:

- List in Dart,
- Set in Dart,
- Map in Dart and
- Where in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

LIST IN DART

List In Dart

If you want to store multiple values in the same variable, you can use **List**. List in dart is similar to **Arrays** in other programming languages. E.g. to store the names of multiple students, you can use a List. The List is represented by **Square Braces[].**

How To Create List

You can create a List by specifying the initial elements in a square bracket. Square bracket **[]** is used to represent a List.

```
// Integer List
List<int> ages = [10, 30, 23];

// String List
List<String> names = ["Raj", "John", "Rocky"];

// Mixed List
var mixed = [10, "John", 18.8];
```

Types Of Lists

- Fixed Length List
- Growable List [**Mostly Used**]

Fixed Length List

The fixed-length Lists are defined with the specified length. You cannot change the size at runtime. This will create List of 5 integers with the value 0.

```
void main() {
   var list = List<int>.filled(5,0);
   print(list);
}
```

Show Output

```
[0, 0, 0, 0, 0]
```

Note: You cannot add a new item to **Fixed Length List**, but you can change the values of List.

Growable List

A List defined without a specified length is called Growable List. The length of the growable List can be changed in runtime.

```
void main() {
   var list1 = [210,21,22,33,44,55];
   print(list1);
}
```

Show Output

```
[210, 21, 22, 33, 44, 55]
```

Access Item Of List

You can access the List item by **index**. Remember that the List index always starts with **0**.

```
void main() {
  var list = [210, 21, 22, 33, 44, 55];

  print(list[0]);
  print(list[1]);
  print(list[2]);
  print(list[3]);
  print(list[4]);
  print(list[5]);
}
```

Show Output

```
210
21
22
33
44
55
```

Get Index By Value

You can also get the index by value.

```
void main() {
  var list = [210, 21, 22, 33, 44, 55];

  print(list.indexOf(22));
  print(list.indexOf(33));
}
```

Show Output

```
2
3
```

Find The Length Of The List

You can find the length of List by using **.length** property.

```
void main(){
   List<String> names = ["Raj", "John", "Rocky"];
   print(names.length);
 }
```

Show Output

```
3
```

Note: Remember that List **index** starts with **0** and length always starts with **1**.

Changing Values Of List

You can also change the value of List. You can do it by **listName[index]=value;**. For more, see the example below.

```
void main(){
   List<String> names = ["Raj", "John", "Rocky"];
   names[1] = "Bill";
   names[2] = "Elon";
   print(names);
}
```

Show Output

```
[Raj, Bill, Elon]
```

Mutable And Immutable List

A mutable List means they can change after the declaration, and an immutable List means they can’t change after the declaration.

```
List<String> names = ["Raj", "John", "Rocky"]; // Mutable List
names[1] = "Bill"; // possible
names[2] = "Elon"; // possible

const List<String> names = ["Raj", "John", "Rocky"]; // Immutable List
names[1] = "Bill"; // not possible
names[2] = "Elon"; // not possible
```

List Properties In Dart

- **first**: It returns the first element in the List.
- **last**: It returns the last element in the List.
- **isEmpty**: It returns **true** if the List is empty and **false** if the List is not empty.
- **isNotEmpty**: It returns **true** if the List is not empty and **false** if the List is empty.
- **length**: It returns the length of the List.
- **reversed**: It returns a List in reverse order.
- **single**: It is used to check if the List has only one element and returns it.

Access First And Last Elements Of List

You can access the first and last elements in the List by:

```
void main() {
   List<String> drinks = ["water", "juice", "milk", "coke"];
   print("First element of the List is: ${drinks.first}");
   print("Last element of the List is: ${drinks.last}");
}
```
Show Output

```
First element of the List is: water
Last element of the List is: coke
```

Check The List Is Empty Or Not

You can also check List contain any elements inside it or not. It will give result either in **true** or in **false**.

```
void main() {
   List<String> drinks = ["water", "juice", "milk", "coke"];
   List<int>  ages = [];
   print("Is drinks Empty: "+drinks.isEmpty.toString());
   print("Is drinks not Empty: "+drinks.isNotEmpty.toString());
   print("Is ages Empty: "+ages.isEmpty.toString());
   print("Is ages not Empty: "+ages.isNotEmpty.toString());

}
```
Show Output

```
Is drinks Empty: false
Is drinks not Empty: true
Is ages Empty: true
Is ages not Empty: false
```

Reverse List In Dart

You can easily reverse List by using **.reversed** properties. Here is an example below:

```
void main() {
   List<String> drinks = ["water", "juice", "milk", "coke"];
   print("List in reverse: ${drinks.reversed}");
}
```
Show Output

```
List in reverse: (coke, milk, juice, water)
```

Adding Item To List

Dart provides four methods to insert the elements into the Lists. These methods are given below.

Method	Description
add()	Add one element at a time and returns the modified List object.
addAll()	Insert the multiple values to the given List, and each value is separated by the commas and enclosed with a square bracket ([]).
insert()	Provides the facility to insert an element at a specified index position.
insertAll()	Insert the multiple value at the specified index position.

Example 1: Add Item To List

In this example below, we are adding an item to evenList using **add()** method.

```
void main() {
    var evenList = [2,4,6,8,10];
    print(evenList);
    evenList.add(12);
    print(evenList);
}
```

Show Output

```
[2, 4, 6, 8, 10]
[2, 4, 6, 8, 10, 12]
```

Example 2: Add Items To List

In this example below, we are adding items to evenList using **addAll()** method.

```
void main() {
  var evenList = [2, 4, 6, 8, 10];
  print(evenList);
  evenList.addAll([12, 14, 16, 18]);
  print(evenList);
}
```

Show Output

```
[2, 4, 6, 8, 10]
[2, 4, 6, 8, 10, 12, 14, 16, 18]
```

Example 3: Insert Item To List

In this example below, we are adding an item to myList using **insert()** method.

```
void main() {
  List myList = [3, 4, 2, 5];
  print(myList);
  myList.insert(2, 15);
  print(myList);
}
```

Show Output

```
[3, 4, 2, 5]
[3, 4, 15, 2, 5]
```

**Example 4: Insert Items To List **

In this example below, we are adding items to myList using **insertAll()** method.

```
void main() {
  var myList = [3, 4, 2, 5];
  print(myList);
  myList.insertAll(1, [6, 7, 10, 9]);
  print(myList);
}
```

Show Output

```
[3, 4, 2, 5]
[3, 6, 7, 10, 9, 4, 2, 5]
```

Replace Range Of List

You can also replace the range of the List. For more, see the example below.

```
void main() {
  var list = [10, 15, 20, 25, 30];
  print("List before updation: ${list}");
  list.replaceRange(0, 4, [5, 6, 7, 8]);
  print("List after updation using replaceAll() function : ${list}");
}
```

Show Output

```
List before updation: [10, 15, 20, 25, 30]
List after updation using replaceAll() function : [5, 6, 7, 8, 30]
```

Removing List Elements

Method	Description
remove()	Removes one element at a time from the given List.
removeAt()	Removes an element from the specified index position and returns it.
removeLast()	Remove the last element from the given List.
removeRange()	Removes the item within the specified range.

Example 1: Removing List Item From List

In this example below, we are removing item of List using **remove()** method.

```
void main() {
  var list = [10, 20, 30, 40, 50];
  print("List before removing element : ${list}");
  list.remove(30);
  print("List after removing element : ${list}");
}
```

Show Output

```
List before removing element : [10, 20, 30, 40, 50]
List after removing element : [10, 20, 40, 50]
```

Example 2: Removing List Item From List

In this example below, we are removing item of List using **removeAt()** method.

```
void main() {
  var list = [10, 11, 12, 13, 14];
  print("List before removing element : ${list}");
  list.removeAt(3);
  print("List after removing element : ${list}");
}
```

Show Output

```
List before removing element : [10, 11, 12, 13, 14]
List after removing element : [10, 11, 12, 14]
```

Example 3: Removing Last Item From List

In this example below, we are removing last item of List using **removeLast()** method.

```dart
void main() {
  var list = [10, 20, 30, 40, 50];
  print("List before removing element:${list}");
  list.removeLast();
  print("List after removing last element:${list}");
}
```

Show Output

```
List before removing element:[10, 20, 30, 40, 50]
List after removing last element:[10, 20, 30, 40]
```

Example 4: Removing List Range From List

In this example below, we are removing the range of items of List using **removeRange()** method.

```dart
void main() {
  var list = [10, 20, 30, 40, 50];
  print("List before removing element:${list}");
  list.removeRange(0, 3);
  print("List after removing range element:${list}");
}
```

Show Output

```
List before removing element:[10, 20, 30, 40, 50]
List after removing range element:[40, 50]
```

Loops In List

You can use for loop, for each loop, or any other type of loop.

```dart
void main() {
  List<int> list = [10, 20, 30, 40, 50];
  list.forEach((n) => print(n));
}
```

Show Output

```
10
20
30
40
50
```

Multiply All Value By 2 Of All List

This example below multiply value of List item by 2.

```
void main() {
  List<int> list = [10, 20, 30, 40, 50];
  var douledList = list.map((n) => n * 2);

  print((douledList));
}
```

Show Output

```
(20, 40, 60, 80, 100)
```

Combine Two Or More List In Dart

You can combine two or more Lists in dart by using **spread** syntax.

```
void main() {
  List<String> names = ["Raj", "John", "Rocky"];
  List<String> names2 = ["Mike", "Subash", "Mark"];

  List<String> allNames = [...names, ...names2];
  print(allNames);
}
```

Show Output

```
[Raj, John, Rocky, Mike, Subash, Mark]
```

Conditions In List

You can also use conditions in List. Here **sad = false** so cart doesn't contain **Beer** in it.

```
void main() {
  bool sad = false;
  var cart = ['milk', 'ghee', if (sad) 'Beer'];
  print(cart);
}
```

Show Output

```
[milk, ghee]
```

Where In List Dart

You can use where with List to filter specific items. Here in this example, even numbers are only filtered.

```
void main(){
```

```
List<int> numbers = [2,4,6,8,10,11,12,13,14];

List<int> even = numbers.where((number)=> number.isEven).toList();
print(even);
}
```
Show Output

```
[2, 4, 6, 8, 10, 12, 14]
```

Note: Choose Lists if order matters. You can easily add items to the end. Searching can be slow when the List size is big.

SET IN DART

Set In Dart

Set is a unique collection of items. You cannot store duplicate values in the Set. It is unordered, so it can be faster than lists while working with a large amount of data. Set is useful when you need to store unique values without considering the order of the input. E.g., fruits name, months name, days name, etc. It is represented by **Curley Braces{}.**

Note: The list allows you to add **duplicate items**, but the Set doesn't allow it.

Syntax

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

How To Create A Set In Dart

You can create a Set in Dart using the **Set** type annotation. Here **Set<String>** means only text is allowed in the Set.

```
void main(){
  Set<String> fruits = {"Apple", "Orange", "Mango"};
  print(fruits);
}
```
Show Output

```
{Apple, Orange, Mango}
```

Set Properties In Dart

Properties	Work
first	To get first value of Set.
last	To get last value of Set.
isEmpty	Return true or false.
isNotEmpty	Return true or false.
length	It returns the length of the Set.

Example of Set Properties Dart

This example finds the first and last element of the Set, checks whether it is empty or not, and finds its length.

```
void main() {
  // declaring fruits as Set
  Set<String> fruits = {"Apple", "Orange", "Mango", "Banana"};

  // using different properties of Set
  print("First Value is ${fruits.first}");
  print("Last Value is ${fruits.last}");
  print("Is fruits empty? ${fruits.isEmpty}");
  print("Is fruits not empty? ${fruits.isNotEmpty}");
  print("The length of fruits is ${fruits.length}");
}
```

Show Output

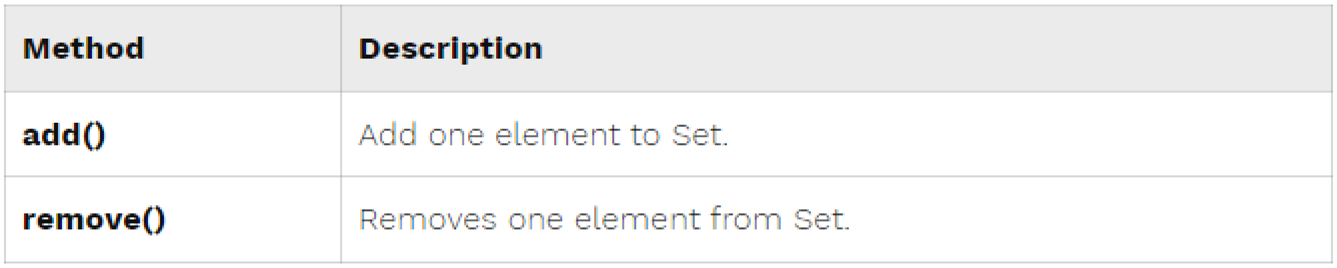
```
First Value is Apple
Last Value is Banana
Is fruits empty? false
Is fruits not empty? true
The length of fruits is 4
```

Check The Available Value

If you want to see whether the Set contains specific items or not, you can use the **contains** method, which returns true or false.

```
void main(){
  Set<String> fruits = {"Apple", "Orange", "Mango"};
  print(fruits.contains("Mango"));
  print(fruits.contains("Lemon"));
}
```

Show Output



Method	Description
add()	Add one element to Set.
remove()	Removes one element from Set.

```
true
false
```

Add & Remove Items In Set

Like lists, you can add or remove items in a Set. To add items use **add()** method and to remove use **remove()** method.

```
void main(){
 Set<String> fruits = {"Apple", "Orange", "Mango"};

  fruits.add("Lemon");
  fruits.add("Grape");

  print("After Adding Lemon and Grape: $fruits");

  fruits.remove("Apple");
  print("After Removing Apple: $fruits");
}
```

Show Output

```
After Adding Lemon and Grape: {Apple, Orange, Mango, Lemon, Grape}
After Removing Apple: {Orange, Mango, Lemon, Grape}
```

Adding Multiple Elements

You can use **addAll()** method to add multiple elements from the list to Set.

Method	**Description**
addAll()	Insert the multiple values to the given Set.

```
void main(){
 Set<int> numbers = {10, 20, 30};
  numbers.addAll([40,50]);
 print("After adding 40 and 50: $numbers");
}
```

Show Output

```
After adding 40 and 50: {10, 20, 30, 40, 50}
```

Printing All Values In Set

You can print all Set items by using loops. Click here if you want to learn loop in dart.

```
void main(){
 Set<String> fruits = {"Apple", "Orange", "Mango"};

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

Show Output

```
Apple
Orange
Mango
```

Set Methods In Dart

Some other helpful Set methods in dart.

Method	Description
clear()	Removes all elements from the Set.
difference()	Creates a new Set with the elements of this that are not in other.
elementAt()	Returns the index value of element.
intersection()	Find common elements in two sets.

Clear Set In Dart

In this example, you can see how to remove all items from the Set in dart.

```
void main() {
  Set<String> fruits = {"Apple", "Orange", "Mango"};
  // to clear all items
  fruits.clear();

  print(fruits);
}
```

Show Output

```
{}
```

Difference In Set

In Dart, the difference method creates a new Set with the elements that are not in the other.

```
void main() {
  Set<String> fruits1 = {"Apple", "Orange", "Mango"};
  Set<String> fruits2 = {"Apple", "Grapes", "Banana"};

  final differenceSet = fruits1.difference(fruits2);

  print(differenceSet);
}
```

Show Output

```
{Orange, Mango}
```

Element At Method In Dart

In Dart you can find the Set value by its index number. The index number starts with 0.

```
void main() {
  Set<String> days = {"Sunday", "Monday", "Tuesday"};
  // index starts from 0 so 2 means Tuesday
  print(days.elementAt(2));
}
```

Show Output

```
Tuesday
```

Intersection Method In Dart

In Dart, the intersection method creates a new Set with the common elements in 2 Sets. Here Apple is available in both Sets.

```
void main() {
  Set<String> fruits1 = {"Apple", "Orange", "Mango"};
  Set<String> fruits2 = {"Apple", "Grapes", "Banana"};

  final intersectionSet = fruits1.intersection(fruits2);

  print(intersectionSet);
}
```

Show Output

```
{Apple}
```

MAP IN DART

Map In Dart

In a Map, data is stored as keys and values. In Map, each key must be unique. They are similar to HashMaps and Dictionaries in other languages.

How To Create Map In Dart

Here we are creating a Map for **String** and **String**. It means keys and values must be the type of String. You can create a Map of any kind as you like.

```
void main(){
Map<String, String> countryCapital = {
  'USA': 'Washington, D.C.',
  'India': 'New Delhi',
  'China': 'Beijing'
};
```

```
  print(countryCapital);
}
```

Show Output

```
{USA: Washington, D.C., India: New Delhi, China: Beijing}
```

Note: Here **Usa**, **India**, and **China** are keys, and it must be **unique**.

Access Value From Key

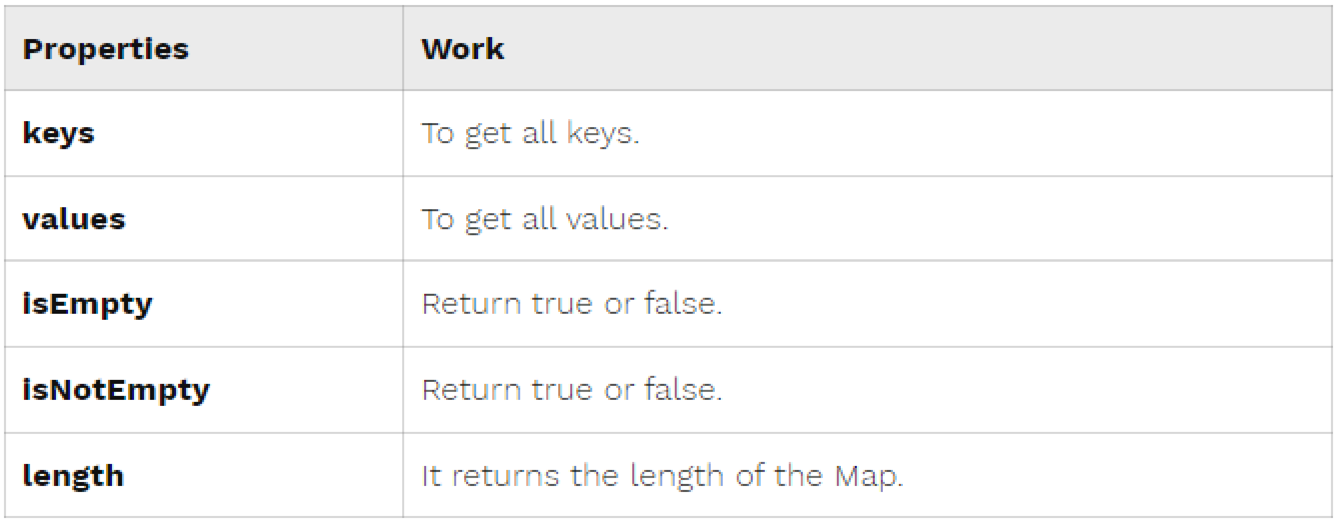
You can find the value of Map from its key. Here we are printing **Washington, D.C.** by its key, i.e., **USA**.

```
void main(){
Map<String, String> countryCapital = {
  'USA': 'Washington, D.C.',
  'India': 'New Delhi',
  'China': 'Beijing'
};
  print(countryCapital["USA"]);
}
```

Show Output

```
Washington, D.C.
```

Map Properties In Dart



Properties	**Work**
keys	To get all keys.
values	To get all values.
isEmpty	Return true or false.
isNotEmpty	Return true or false.
length	It returns the length of the Map.

Example Of Map Properties In Dart

This example finds all keys/values of Map, the first and last element, checks whether it is empty or not, and finds its length.

```dart
void main() {

  Map<String, double> expenses = {
    'sun': 3000.0,
    'mon': 3000.0,
    'tue': 3234.0,
  };

  print("All keys of Map: ${expenses.keys}");
  print("All values of Map: ${expenses.values}");
  print("Is Map empty: ${expenses.isEmpty}");
  print("Is Map not empty: ${expenses.isNotEmpty}");
  print("Length of map is: ${expenses.length}");
}
```

Show Output

```
All keys of Map: (sun, mon, tue)
All values of Map: (3000, 3000, 3234)
Is Map empty: false
Is Map empty: true
Length of map is: 3
```

Adding Element To Map

If you want to add an element to the existing Map. Here is the way for you:

```dart
void main(){
Map<String, String> countryCapital = {
  'USA': 'Washington, D.C.',
  'India': 'New Delhi',
  'China': 'Beijing'
};
  // Adding New Item
  countryCapital['Japan'] = 'Tokio';
  print(countryCapital);
}
```

Show Output

```
{USA: Washington, D.C., India: New Delhi, China: Beijing, Japan: Tokio}
```

Updating An Element Of Map

If you want to update an element of the existing Map. Here is the way for you:

```dart
void main(){
Map<String, String> countryCapital = {
  'USA': 'Nothing',
  'India': 'New Delhi',
  'China': 'Beijing'
```

Properties	Work
keys.toList()	Convert all Maps keys to List.
values.toList()	Convert all Maps values to List.
containsKey('key')	Return true or false.
containsValue('value')	Return true or false.
clear()	Removes all elements from the Map.
removeWhere()	Removes all elements from the Map if condition is valid.

```
};
  // Updating Item
  countryCapital['USA'] = 'Washington, D.C.';
  print(countryCapital);
}
```

Show Output

```
{USA: Washington, D.C., India: New Delhi, China: Beijing}
```

Map Methods In Dart

Some useful Map methods in dart.

Convert Maps Keys & Values To List

Let's convert keys and values of Map to List.

```
void main() {

  Map<String, double> expenses = {
    'sun': 3000.0,
    'mon': 3000.0,
    'tue': 3234.0,
  };

  // Without List
  print("All keys of Map: ${expenses.keys}");
  print("All values of Map: ${expenses.values}");

  // With List
  print("All keys of Map with List: ${expenses.keys.toList()}");
  print("All values of Map with List: ${expenses.values.toList()}");

}
```
Show Output

```
All keys of Map: (sun, mon, tue)
All values of Map: (3000, 3000, 3234)
All keys of Map with List: [sun, mon, tue]
All values of Map with List: [3000, 3000, 3234]
```

Check Map Contains Specific Key/Value Or Not?

Let’s check whether the Map contains a specific key/value in it or not.

```
void main() {

  Map<String, double> expenses = {
    'sun': 3000.0,
    'mon': 3000.0,
    'tue': 3234.0,
  };

  // For Keys
  print("Does Map contain key sun: ${expenses.containsKey("sun")}");
  print("Does Map contain key abc: ${expenses.containsKey("abc")}");

  // For Values
  print("Does Map contain value 3000.0:
${expenses.containsValue(3000.0)}");
  print("Does Map contain value 100.0: ${expenses.containsValue(100.0)}");

}
```
Show Output

```
Does Map contain key sun: true
Does Map contain key abc: false
Does Map contain value 3000.0: true
Does Map contain value 100.0: false
```

Removing Items From Map

Suppose you want to remove an element of the existing Map. Here is the way for you:

```
void main(){
```

```
Map<String, String> countryCapital = {
  'USA': 'Nothing',
  'India': 'New Delhi',
  'China': 'Beijing'
};

  countryCapital.remove("USA");
  print(countryCapital);
}
```

Show Output

```
{India: New Delhi, China: Beijing}
```

Looping Over Element Of Map

You can use any loop in Map to print all keys/values or to perform operations in its keys and values.

```
void main(){

  Map<String, dynamic> book = {
    'title': 'Misson Mangal',
    'author': 'Kuber Singh',
    'page': 233
  };

 // Loop Through Map
  for(MapEntry book in book.entries){
    print('Key is ${book.key}, value ${book.value}');
  }
}
```

Show Output

```
Key is title, value Misson Mangal
Key is author, value Kuber Singh
Key is page, value 233
```

Looping In Map Using For Each

In this example, you will see how to use a loop to print all the keys and values in Map.

```
void main(){

  Map<String, dynamic> book = {
    'title': 'Misson Mangal',
    'author': 'Kuber Singh',
    'page': 233
  };
```

```


 // Loop Through For Each
  book.forEach((key,value)=> print('Key is $key and value is $value'));

}
```

Show Output

```
Key is title and value is Misson Mangal
Key is author and value is Kuber Singh
Key is page and value is 233
```

Remove Where In Dart Map

In this example, you will see how to get students whose marks are greater or equal to 32 using where method.

```
void main() {
  Map<String, double> mathMarks = {
    "ram": 30,
    "mark": 32,
    "harry": 88,
    "raj": 69,
    "john": 15,
  };
  mathMarks.removeWhere((key, value) => value < 32);
  print(mathMarks);
}
```

Show Output

```
{mark: 32.0, harry: 88.0, raj: 69.0}
```

WHERE IN DART

Where Dart

You can use where in list, set, map to **filter specific items**. It returns a new list containing all the elements that satisfy the condition. This is also called **Where Filter** in dart. Let’s see the syntax below:

Syntax

```
Iterable<E> where(
bool test(
E element
)
)
```

Example 1: Filter Only Odd Number From List

In this example, you will get only odd numbers from a list.

```
void main() {
  List<int> numbers = [2, 4, 6, 8, 10, 11, 12, 13, 14];

  List<int> oddNumbers = numbers.where((number) => number.isOdd).toList();
  print(oddNumbers);
}
```

Show Output

```
[11, 13]
```

Example 2: Filter Days Start With S

In this example, you will get only days that start with alphabet s.

```
void main() {
  List<String> days = [
    "Sunday",
    "Monday",
    "Tuesday",
    "Wednesday",
    "Thursday",
    "Friday",
    "Saturday"
  ];

  List<String> startWithS =
      days.where((element) => element.startsWith("S")).toList();

  print(startWithS);
}
```

Show Output

```
[Sunday, Saturday]
```

Example 3: Where Filter In Map

In this example, you will get students whose marks are greater or equal to 32.

```
void main() {
  Map<String, double> mathMarks = {
    "ram": 30,
    "mark": 32,
    "harry": 88,
    "raj": 69,
    "john": 15,
  };

  mathMarks.removeWhere((key, value) => value < 32);

  print(mathMarks);
}
```

Show Output

```
{mark: 32.0, harry: 88.0, raj: 69.0}
```

QUESTION FOR PRACTICE 4

Question For Practice 4

1. Create a list of names and print all names using list.
2. Create a set of fruits and print all fruits using loop.
3. Create a program thats reads list of expenses amount using user input and print total.
4. Create an empty list of type string called days. Use the add method to add names of 7 days and print all days.
5. Add your 7 friend names to the list. Use where to find a name that starts with alphabet a.
6. Create a map with name, address, age, country keys and store values to it. Update country name to other country and print all keys and values.
7. Create a map with name, phone keys and store some values to it. Use where to find all keys that have length 4.
8. Create a simple to-do application that allows user to add, remove, and view their task.

Dart File Handling

This section will help you to handle files in Dart. File handling is an important part of any programming language. Here you will learn the following topics:

- Read File in Dart,
- Write File in Dart,
- Delete File in Dart,

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

READ FILE IN DART

Introduction To File Handling

File handling is an important part of any programming language. In this section, you will learn how to read the file in a dart programming language.

Read File In Dart

Assume that you have a file named `test.txt` in the same directory of your dart program.

```
Welcome to test.txt file.
This is a test file.
```

Now, you can read this file using **File** class and **readAsStringSync()** method.

```
// dart program to read from file
import 'dart:io';

void main() {
  // creating file object
  File file = File('test.txt');
  // read file
  String contents = file.readAsStringSync();
  // print file
  print(contents);
}
```

Show Output

```
Welcome to test.txt file.
This is a test file.
```

Get File Information

In this example below, you will learn how to get file information like file location, file size, and last modified time.

```
import 'dart:io';

void main() {
  // open file
  File file = File('test.txt');
  // get file location
  print('File path: ${file.path}');
  // get absolute path
  print('File absolute path: ${file.absolute.path}');
  // get file size
  print('File size: ${file.lengthSync()} bytes');
  // get last modified time
  print('Last modified: ${file.lastModifiedSync()}');
}
```

Show Output

```
File path: test.txt
File absolute path: /home/iambrp/Desktop/Dart Practice/test.txt
File size: 25 bytes
Last modified: 2023-01-28 11:00:32.000
```

Note: If you try to get information of a file that does not exist, then it will throw an exception.

CSV File

A CSV (**Comma Separated Values**) file is a plain text file that contains data organized in a table format, where columns are separated by commas and rows are separated by line breaks. CSV files are used for:

- Data exchange between different applications.
- Data backup and restore.
- Importing and exporting data from databases.
- Automation of data processing tasks.

Read CSV File In Dart

Assume that you have a CSV file named `test.csv` in the same directory of your dart program.

```
Name,Email,Phone
John, john@gmail.com, 1234567890
Smith, smith@gmail.com, 0987654321
```

Now, you can read this file using **File** class and **readAsStringSync()** method. We will use **split()** method to split the string into a list of strings.

```
// dart program to read from csv file
import 'dart:io';

void main() {
  // open file
  File file = File('test.csv');
  // read file
  String contents = file.readAsStringSync();
  // split file using new line
  List<String> lines = contents.split('\n');
  // print file
  print('----------------------');
  for (var line in lines) {
    print(line);
  }
}
```

Show Output

```
---------------------
Name,Email,Phone
John, john@gmail.com, 1234567890
Smith, smith@gmail.com, 0987654321
```

Read Only Part Of File

You can read only part of file using **substring()** method. Here is an example to read only first 10 characters of file. Make sure that you have a file named **test.txt** in the same directory of your dart program.

```
Welcome to test.txt file
This is a test file.
// dart program to read from file
import 'dart:io';

void main() {
  // open file
  File file = new File('test.txt');
  // read only first 10 characters
  String contents = file.readAsStringSync().substring(0, 10);
  // print file
  print(contents);
}
```

Show Output

```
Welcome to
```

Read File From Specific Directory

To read a file from a specific directory, you need to provide the full path of the file. Here is an example to read file from a specific directory.

```
// dart program to read from file
import 'dart:io';

void main() {
  // open file
  File file = File('C:\\Users\\test.txt');
  // read file
  String contents = file.readAsStringSync();
  // print file
  print(contents);
}
```

Show Output

```
Welcome to test.txt file.
This is a test file.
```

WRITE FILE IN DART

Introduction

In this section, you will learn how to write file in dart programming language by using **File** class and **writeAsStringSync()** method.

Write File In Dart

Let’s create a file named **test.txt** in the same directory of your dart program and write some text in it.

```
// dart program to write to file
import 'dart:io';

void main() {
  // open file
  File file = File('test.txt');
  // write to file
  file.writeAsStringSync('Welcome to test.txt file.');
  print('File written.');
}
```

Show Output

```
File written.
```

Note: If you have already some content in **test.txt** file, then it will be removed and replaced with new content.

Add New Content To Previous Content

You can use **FileMode.append** to add new content to previous content. Assume that **test.txt** file already contains some text.

```
Welcome to test.txt file.
```

Now, let’s add new content to it.

```
// dart program to write to existing file
import 'dart:io';

void main() {
  // open file
  File file =  File('test.txt');
  // write to file
  file.writeAsStringSync('\nThis is a new content.', mode:
FileMode.append);
```

```
  print('Congratulations!! New content is added on top of previous content.');
}
```

Show Output

```
Congratulations!! New content is added on top of previous content.
```

Write CSV File In Dart

In the example below, we will ask user to enter **name** and **phone** of 3 students and write it to a csv file named **students.csv**.

```
// dart program to write to csv file
import 'dart:io';

void main() {
  // open file
  File file = File("students.csv");
  // write to file
  file.writeAsStringSync('Name,Phone\n');
  for (int i = 0; i < 3; i++) {
    // user input name
    stdout.write("Enter name of student ${i + 1}: ");
    String? name = stdin.readLineSync();
    stdout.write("Enter phone of student ${i + 1}: ");
    // user input phone
    String? phone = stdin.readLineSync();
    file.writeAsStringSync('$name,$phone\n', mode: FileMode.append);
  }
  print("Congratulations!! CSV file written successfully.");
}
```

Show Output

```
Enter name of student 1: John
Enter phone of student 1: 1234567890
Enter name of student 2: Mark
Enter phone of student 2: 0123456789
Enter name of student 3: Elon
Enter phone of student 3: 0122112322
Congratulations!! CSV file written successfully.
```

students.csv file will look like this:

```
Name,Phone
John,1234567890
Mark,0123456789
Elon,0122112322
```

Note: You can create any type of file using **writeAsStringSync()** method. For example, **.html**, **.json**, **.xml**, etc.

DELETE FILE IN DART

Introduction

In this section, you will learn how to delete file in dart programming language using **File** class and **deleteSync()** method.

Delete File In Dart

Assume that you have a file named **test.txt** in the same directory of your dart program. Now, let's delete it.

```
// dart program to delete file
import 'dart:io';

void main() {
  // open file
  File file = File('test.txt');
  // delete file
  file.deleteSync();
  print('File deleted.');
}
```

Show Output

```
File deleted.
```

Note: If you try to delete a file that does not exist, then it will throw an exception.

Delete File If Exists

You can use **File.existsSync()** method to check if a file exists or not. If it exists, then you can delete it.

```
// dart program to delete file if exists
import 'dart:io';

void main() {
  // open file
  File file = File('test.txt');
  // check if file exists
  if (file.existsSync()) {
    // delete file
    file.deleteSync();
    print('File deleted.');
  } else {
    print('File does not exist.');
  }
}
```

Show Output

```
File does not exist.
```

QUESTION FOR PRACTICE 5

Dart File Handling Practice Questions

1. Write a dart program to add your name to “hello.txt” file.
2. Write a dart program to append your friends name to a file that already has your name.
3. Write a dart program to get the current working directory.
4. Write a dart program to copy the “hello.txt” file to “hello_copy.txt” file.
5. Write a dart program to create 100 files using loop.
6. Write a dart program to delete the file “hello_copy.txt”. Make sure you have created the file “hello_copy.txt.
7. Write a dart program to store name, age, and address of students in a csv file and read it.

Dart Object Oriented Programming

This section will teach you the basics of Dart OOP so that you can start creating amazing programs right away. Here you will learn the following topics:

- OOP in Dart,
- Class in Dart,
- Objects in Dart,
- Class and Objects in Dart,
- Constructor in Dart,
- Default Constructor in Dart,
- Parameterized Constructor in Dart,
- Named Constructor in Dart,
- Constant Constructor in Dart,
- Encapsulation in Dart,
- Getter in Dart,
- Setter in Dart,
- Getter and Setter in Dart,
- Inheritance in Dart,
- Inheritance of Constructors in Dart,
- Super Keyword in Dart,
- Polymorphism in Dart,
- Static in Dart,
- Enum in Dart,
- Abstract Class in Dart,
- Interface in Dart.
- Mixin in Dart,

- Factory Constructor in Dart, and
- Generics in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

Accelerate your Workflow

Save any snippet in this tutorial to your personal micro-repository with Pieces for Developers to speed up your workflow. Pieces is a centralized productivity suite that leverages AI to help developers save snippets, extract code from screenshots, auto-enrich code, and much more.

OOP In Dart

Object-oriented programming (OOP) is a programming method that uses objects and their interactions to design and program applications. It is one of the most popular programming paradigms and is used in many programming languages, such as Dart, Java, C++, Python, etc.

In **OOP**, an object can be anything, such as a person, a bank account, a car, or a house. Each object has its attributes (or properties) and behavior (or methods). For example, a person object may have the attributes **name**, **age** and **height**, and the behavior **walk** and **talk**.

Advantages

- It is easy to understand and use.
- It increases reusability and decreases complexity.
- The productivity of programmers increases.
- It makes the code easier to maintain, modify and debug.
- It promotes teamwork and collaboration.
- It reduces the repetition of code.

Features Of OOP

1. Class
2. Object
3. Encapsulation
4. Inheritance
5. Polymorphism
6. Abstraction

Note: The main purpose of OOP is to break complex problems into smaller objects. You will learn all these OOPs features later in this dart tutorial.

Key Points

- Object Oriented Programming (OOP) is a programming paradigm that uses objects and their interactions to design and program applications.
- OOP is based on objects, which are data structures containing data and methods.
- OOP is a way of thinking about programming that differs from traditional procedural programming.
- OOP can make code more modular, flexible, and extensible.
- OOP can help you to understand better and solve problems.

Class In Dart

In object-oriented programming, a class is a blueprint for creating objects. A class defines the properties and methods that an object will have. For example, a class called **Dog** might have properties like **breed**, **color** and methods like **bark**, **run**.

Declaring Class In Dart

You can declare a class in dart using the **class** keyword followed by class name and braces {}. It's a good habit to write class name in **PascalCase**. For example, **Employee**, **Student**, **QuizBrain**, etc.

Syntax

```
class ClassName {
// properties or fields
// methods or functions
}
```

In the above syntax:

- The **class** keyword is used for defining the class.
- **ClassName** is the name of the class and must start with capital letter.
- Body of the class consists of **properties** and **functions**.
- **Properties** are used to store the data. It is also known as **fields** or **attributes**.
- **Functions** are used to perform the operations. It is also known as **methods**.

Example 1: Declaring A Class In Dart

In this example below, there is class **Animal** with three properties: **name**, **numberOfLegs**, and **lifeSpan**. The class also has a method called **display**, which prints out the values of the three properties.

```
class Animal {
  String? name;
  int? numberOfLegs;
  int? lifeSpan;

  void display() {
    print("Animal name: $name.");
    print("Number of Legs: $numberOfLegs.");
    print("Life Span: $lifeSpan.");
  }
}
```

Note: This program will not print anything because we have not created any object of the class. You will learn about the object later. The **?** is used for null safety. You will also learn about null safety later.

Example 2: Declaring A Person Class In Dart

In this example below, there is class **Person** with four properties: **name**, **phone**, **isMarried**, and **age**. The class also has a method called **displayInfo**, which prints out the values of the four properties.

```
class Person {
  String? name;
  String? phone;
  bool? isMarried;
  int? age;

  void displayInfo() {
    print("Person name: $name.");
    print("Phone number: $phone.");
    print("Married: $isMarried.");
    print("Age: $age.");
  }
}
```

Example 3: Declaring Area Class In Dart

In this example below, there is class **Area** with two properties: **length** and **breadth**. The class also has a method called **calculateArea**, which calculates the area of the rectangle.

```
class Area {
  double? length;
  double? breadth;

  double calculateArea() {
    return length! * breadth!;
  }
}
```

Example 4: Declaring A Student Class In Dart

In this example below, there is class **Student** with three properties: **name**, **age**, and **grade**. The class also has a method called **displayInfo**, which prints out the values of the three properties.

```
class Student {
  String? name;
  int? age;
  int? grade;

  void displayInfo() {
    print("Student name: $name.");
    print("Student age: $age.");
    print("Student grade: $grade.");
  }
}
```

Key Points

- The class is declared using the **class** keyword.
- The class is a blueprint for creating objects.
- The class body consists of properties and methods.
- The properties are also known as fields, attributes, or data members.
- The methods are also known as behaviors, or member functions.

Challenge

Create a class **Book** with three properties: **name**, **author**, and **price**. Also, create a method called **display**, which prints out the values of the three properties.

Note: In the next section, you will learn how to create an object from a class.

Object In Dart

In object-oriented programming, an object is a self-contained unit of code and data. Objects are created from templates called classes. An object is made up of properties(variables) and methods(functions). An object is an instance of a class.

For example, a bicycle object might have attributes like color, size, and current speed. It might have methods like changeGear, pedalFaster, and brake.

Note: To create an object, you must create a class first. It's a good practice to declare the object name in lower case.

Instantiation

In object-oriented programming, instantiation is the process of creating an instance of a class. In other words, you can say that instantiation is the process of creating an object of a class. For example, if you have a class called **Bicycle**, then you can create an object of the class called **bicycle**.

Declaring Object In Dart

Once you have created a class, it’s time to declare the object. You can declare an object by the following syntax:

Syntax

```
ClassName objectName = ClassName();
```

Example 1: Declaring An Object In Dart

In this example below, there is class **Bycycle** with three properties: **color**, **size**, and **currentSpeed**. The class has two methods. One is **changeGear**, which changes the gear of the bicycle, and **display** method prints out the values of the three properties. We also have an object of the class **Bycycle** called **bicycle**.

```
    class Bicycle {
      String? color;
      int? size;
      int? currentSpeed;

      void changeGear(int newValue) {
        currentSpeed = newValue;
      }

      void display() {
        print("Color: $color");
        print("Size: $size");
        print("Current Speed: $currentSpeed");
      }
    }

    void main(){
        // Here bicycle is object of class Bicycle.
        Bicycle bicycle = Bicycle();
        bicycle.color = "Red";
        bicycle.size = 26;
        bicycle.currentSpeed = 0;
        bicycle.changeGear(5);
        bicycle.display();
    }
```

Show Output

```
Color: Red
Size: 26
Current Speed: 5
```

Note: Once you create an object, you can access the properties and methods of the object using the dot(.) operator.

Example 2: Declaring Animal Class Object In Dart

In this example below there is class **Animal** with three properties: **name**, **numberOfLegs**, and **lifeSpan**. The class also has a method called **display**, which prints out the values of the three properties. We also have an object of the class **Animal** called **animal**.

```
class Animal {
      String? name;
      int? numberOfLegs;
      int? lifeSpan;

      void display() {
        print("Animal name: $name.");
        print("Number of Legs: $numberOfLegs.");
        print("Life Span: $lifeSpan.");
      }
    }

    void main(){
        // Here animal is object of class Animal.
        Animal animal = Animal();
        animal.name = "Lion";
        animal.numberOfLegs = 4;
        animal.lifeSpan = 10;
        animal.display();
    }
```

Show Output

```
Animal name: Lion.
Number of Legs: 4.
Life Span: 10.
```

Example 3: Declaring Car Class Object In Dart

In this example below there is class **Car** with three properties: **name**, **color**, and **numberOfSeats**. The class also has a method called **start**, which prints out the message “Car Started”. We also have an object of the class **Car** called **car**.

```
class Car {
  String? name;
  String? color;
  int? numberOfSeats;

  void start() {
    print("$name Car Started.");
  }
}

void main(){
    // Here car is object of class Car.
    Car car = Car();
    car.name = "BMW";
    car.color = "Red";
    car.numberOfSeats = 4;
    car.start();

    // Here car2 is another object of class Car.
    Car car2 = Car();
    car2.name = "Audi";
    car2.color = "Black";
    car2.numberOfSeats = 4;
    car2.start();
}
```

Show Output

```
BMW Car Started.
Audi Car Started.
```

Key Points

- The main method is the program's entry point, so it is always needed to see the result.
- The **new** keyword can be used to create a new object, but it is unnecessary.

Challenge

Create a class Camera with properties: **name**, **color**, **megapixel**. Create a method called **display** which prints out the values of the three properties. Create two objects of the class Camera and call the method display.

CLASS AND OBJECTS IN DART

What is Class

A class is a blueprint for creating objects. A class defines the properties and methods that an object will have. If you want to learn more about class in Dart, you can read class in dart.

What is Object

An object is an instance of a class. You can create multiple objects of the same class. If you want to learn more about an object in Dart, you can read object in dart.

Example Of A Class & Object In Dart

In this example below there is class **Animal** with three properties: **name**, **numberOfLegs**, and **lifeSpan**. The class also has a method called **display**, which prints out the values of the three properties.

```dart
class Animal {
  String? name;
  int? numberOfLegs;
  int? lifeSpan;

  void display() {
    print("Animal name: $name.");
    print("Number of Legs: $numberOfLegs.");
    print("Life Span: $lifeSpan.");
  }
}

void main(){
  // Here animal is object of class Animal.
  Animal animal = Animal();
  animal.name = "Lion";
  animal.numberOfLegs = 4;
  animal.lifeSpan = 10;
  animal.display();
}
```

Show Output

```
Animal name: Lion.
Number of Legs: 4.
Life Span: 10.
```

Example 2: Find Area Of Ractangle Using Class and Objects

In this example below there is class **Rectangle** with two properties: **length** and **breadth**. The class also has a method called **area**, which calculates the area of the rectangle.

```dart
class Rectangle{
  //properties of rectangle
  double? length;
  double? breadth;

  //functions of rectangle
  double area(){
    return length! * breadth!;
  }
}

void main(){
  //object of rectangle created
  Rectangle rectangle = Rectangle();

  //setting properties for rectangle
  rectangle.length=10;
  rectangle.breadth=5;

  //functions of rectangle called
  print("Area of rectangle is ${rectangle.area()}.");
}
```

Show Output

```
Area of rectangle is 50.
```

Note: Here **!** is used to tell the compiler that the variable is not null. If you don't use **!**, then you will get an error. You will learn more about it in null safety later.

Example 3: Find Simple Interest Using Class and Objects

In this example below there is class **SimpleInterest** with three properties: **principal**, **rate**, and **time**. The class also has a method called **interest**, which calculates the simple interest.

```
class SimpleInterest{
  //properties of simple interest
  double? principal;
  double? rate;
  double? time;

  //functions of simple interest
  double interest(){
    return (principal! * rate! * time!)/100;
  }
}
void main(){
  //object of simple interest created
  SimpleInterest simpleInterest = SimpleInterest();

  //setting properties for simple interest
  simpleInterest.principal=1000;
  simpleInterest.rate=10;
  simpleInterest.time=2;

  //functions of simple interest called
  print("Simple Interest is ${simpleInterest.interest()}.");
}
```

Show Output

```
Simple Interest is 200.
```

Challenge

Create class Home with properties **name**, **address**, **numberOfRooms**. Create a method called **display** which prints out the values of the properties. Create an object of the class **Home** and set the values of the properties. Call the method **display** to print out the values of the properties.

CONSTRUCTOR IN DART

Introduction

In this section, you will learn about constructor in Dart programming language and how to use constructors with the help of examples. Before learning about the constructor, you should have a basic understanding of the class and object in dart.

Constructor In Dart

A constructor is a special method used to initialize an object. It is called automatically when an object is created, and it can be used to set the initial values for the object's properties. For example, the following code creates a **Person** class object and sets the initial values for the **name** and **age** properties.

```
Person person = Person("John", 30);
```

Without Constructor

If you don't define a constructor for class, then you need to set the values of the properties manually. For example, the following code creates a **Person** class object and sets the values for the **name** and **age** properties.

```
Person person = Person();
person.name = "John";
person.age = 30;
```

Things To Remember

- The constructor's name should be the same as the class name.
- Constructor doesn't have any return type.

Syntax

```
class ClassName {
  // Constructor declaration: Same as class name
  ClassName() {
    // body of the constructor
  }
}
```

Note: When you create a object of a class, the constructor is called automatically. It is used to initialize the values when an object is created.

Example 1: How To Declare Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also created an object of the class **Student** called **student**.

```
class Student {
  String? name;
  int? age;
  int? rollNumber;

  // Constructor
  Student(String name, int age, int rollNumber) {
    print(
        "Constructor called"); // this is for checking the constructor is
called or not.
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}

void main() {
  // Here student is object of class Student.
  Student student = Student("John", 20, 1);
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Constructor called
Name: John
Age: 20
Roll Number: 1
```

Note: The **this** keyword is used to refer to the current instance of the class. It is used to access the current class properties. In the example above, parameter names and class properties of constructor **Student** are the same. Hence to avoid confusion, we use the **this** keyword.

Example 2: Constructor In Dart

In this example below, there is a class **Teacher** with four properties: **name**, **age**, **subject**, and **salary**. Class has one constructor for initializing the values of the properties. Class also contain method **display()** which is used to display the values of the properties. We also created 2 objects of the class **Teacher** called **teacher1** and **teacher2**.

```
class Teacher {
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor
  Teacher(String name, int age, String subject, double salary) {
    this.name = name;
    this.age = age;
    this.subject = subject;
    this.salary = salary;
  }
  // Method
  void display() {
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}\n"); // \n is used for new line
  }
}

void main() {
  // Creating teacher1 object of class Teacher
  Teacher teacher1 = Teacher("John", 30, "Maths", 50000.0);
  teacher1.display();

  // Creating teacher2 object of class Teacher
  Teacher teacher2 = Teacher("Smith", 35, "Science", 60000.0);
  teacher2.display();
}
```

Show Output

```
Name: John
```

```
Age: 30
Subject: Maths
Salary: 50000

Name: Smith
Age: 35
Subject: Science
Salary: 60000
```

Note: You can create many objects of a class. Each object will have its own copy of the properties.

Example 3: Constructor In Dart

In this example below, there is a class **Car** with two properties: **name** and **price**. The class has one constructor for initializing the values of the properties. The class also contains method **display()**, which is used to display the values of the properties. We also created an object of the class **Car** called **car**.

```
 class Car {
  String? name;
  double? price;

  // Constructor
  Car(String name, double price) {
    this.name = name;
    this.price = price;
  }

  // Method
  void display() {
    print("Name: ${this.name}");
    print("Price: ${this.price}");
  }
}

void main() {
  // Here car is object of class Car.
  Car car = Car("BMW", 500000.0);
  car.display();
}
```

Show Output

```
Name: BMW
Price: 500000
```

Example 4: Constructor In Dart

In this example below, there is a class **Staff** with four properties: **name**, **phone1**, **phone2**, and **subject** and one method **display()**. Class has one constructor for initializing the values of only **name**, **phone1** and **subject**. We also created an object of the class **Staff** called **staff**.

```dart
 class Staff {
  String? name;
  int? phone1;
  int? phone2;
  String? subject;

  // Constructor
  Staff(String name, int phone1, String subject) {
    this.name = name;
    this.phone1 = phone1;
    this.subject = subject;
  }

  // Method
  void display() {
    print("Name: ${this.name}");
    print("Phone1: ${this.phone1}");
    print("Phone2: ${this.phone2}");
    print("Subject: ${this.subject}");
  }
}

void main() {
  // Here staff is object of class Staff.
  Staff staff = Staff("John", 1234567890, "Maths");
  staff.display();
}
```

Show Output

```
Name: John
Phone1: 1234567890
Phone2: null
Subject: Maths
```

Example 5: Write Constructor Single Line

In the avobe section, you have written the constructor in long form. You can also write the constructor in short form. You can directly assign the values to the properties. For example, the following code is the short form of the constructor in one line.

```dart
class Person{
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor in short form
  Person(this.name, this.age, this.subject, this.salary);

  // display method
  void display(){
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}");
  }
}

void main(){
  Person person = Person("John", 30, "Maths", 50000.0);
  person.display();
}
```

Show Output

```
Name: John
Age: 30
Subject: Maths
Salary: 50000
```

Example 6: Constructor With Optional Parameters

In the example below, we have created a class **Employee** with four properties: **name**, **age**, **subject**, and **salary**. Class has one constructor for initializing the all properties values. For **subject** and **salary**, we have used optional parameters. It means we can pass or not pass the values of **subject** and **salary**. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Employee** called **employee**.

```
class Employee {
  String? name;
  int? age;
  String? subject;
  double? salary;

  // Constructor
  Employee(this.name, this.age, [this.subject = "N/A", this.salary=0]);

  // Method
  void display() {
    print("Name: ${this.name}");
    print("Age: ${this.age}");
    print("Subject: ${this.subject}");
    print("Salary: ${this.salary}");
  }
}

void main(){
  Employee employee = Employee("John", 30);
  employee.display();
}
```

Show Output

```
Name: John
Age: 30
Subject: N/A
Salary: 0
```

Example 7: Constructor With Named Parameters

In the example below, we have created a class **Chair** with two properties: **name** and **color**. Class has one constructor for initializing the all properties values with named parameters. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Chair** called **chair**.

```
class Chair {
String? name;
String? color;

// Constructor
Chair({this.name, this.color});

// Method
void display() {
  print("Name: ${this.name}");
  print("Color: ${this.color}");
}
}

void main(){
Chair chair = Chair(name: "Chair1", color: "Red");
chair.display();
}
```

Show Output

```
Name: Chair1
Color: Red
```

Example 8: Constructor With Default Values

In the example below, we have created a class **Table** with two properties: **name** and **color**. Class has one constructor for initializing the all properties values with default values. The Class also contain method **display()** which is used to display the values of the properties. We also created an object of the class **Table** called **table**.

```
class Table {
  String? name;
  String? color;

  // Constructor
  Table({this.name = "Table1", this.color = "White"});

  // Method
  void display() {
    print("Name: ${this.name}");
    print("Color: ${this.color}");
  }
}

void main(){
  Table table = Table();
  table.display();
}
```

Show Output

```
Name: Table1
Color: White
```

Key Points

- The constructor's name should be the same as the class name.
- Constructor doesn't have any return type.
- Constructor is only called once at the time of the object creation.
- Constructor is called automatically when an object is created.
- Constructor is used to initialize the values of the properties of the class.

Challenge

Create a class **Patient** with three properties **name**, **age**, and **disease**. The class has one constructor. The constructor is used to initialize the values of the three properties. Also, create an object of the class **Patient** called **patient**. Print the values of the three properties using the object.

DEFAULT CONSTRUCTOR IN DART

Default Constructor

The constructor which is automatically created by the dart compiler if you don't create a constructor is called a default constructor. A default constructor has no parameters. A default constructor is declared using the class name followed by parentheses ().

Example 1: Default Constructor In Dart

In this example below, there is a class **Laptop** with two properties: **brand**, and **price**. Lets create constructor with no parameter and print something from the constructor. We also have an object of the class **Laptop** called **laptop**.

```
class Laptop {
  String? brand;
  int? price;

  // Constructor
  Laptop() {
    print("This is a default constructor");
  }
}

void main() {
  // Here laptop is object of class Laptop.
  Laptop laptop = Laptop();
}
```

Show Output

```
This is a default constructor
```

Note: The default constructor is called automatically when you create an object of the class. It is used to initialize the instance variables of the class.

Example 2: Default Constructor In Dart

In this example below, there is a class **Student** with four properties: **name**, **age**, **schoolname** and **grade**. The default constructor is used to initialize the values of the school name. The reason for this is that the school name is the same for all the students. We also have an object of the class **Student** called **student**. The default constructor is called automatically when you create an object of the class.

```
class Student {
  String? name;
  int? age;
  String? schoolname;
  String? grade;

  // Default Constructor
  Student() {
    print(
        "Constructor called"); // this is for checking the constructor is 
called or not.
    schoolname = "ABC School";
  }
}

void main() {
  // Here student is object of class Student.
  Student student = Student();
  student.name = "John";
  student.age = 10;
  student.grade = "A";
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("School Name: ${student.schoolname}");
  print("Grade: ${student.grade}");
}
```

Show Output

```
Constructor called
Name: John
Age: 10
School Name: ABC School
Grade: A
```

Challenge

Try to create a class **Person** with two properties: **name**, and **planet**. Create a default constructor to initialize the values of the **planet** to earth. Create an object of the class **Person**, set the name to “Your Name” and print the name and planet.

PARAMETERIZED CONSTRUCTOR IN DART

Parameterized Constructor

Parameterized constructor is used to initialize the instance variables of the class. Parameterized constructor is the constructor that takes parameters. It is used to pass the values to the constructor at the time of object creation.

Syntax

```
class ClassName {
  // Instance Variables
```

```dart
  int? number;
  String? name;
  // Parameterized Constructor
  ClassName(this.number, this.name);
}
```

Example 1: Parameterized Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```dart
class Student {
  String? name;
  int? age;
  int? rollNumber;
  // Constructor
  Student(this.name, this.age, this.rollNumber);
}

void main(){
    // Here student is object of class Student.
    Student student = Student("John", 20, 1);
    print("Name: ${student.name}");
    print("Age: ${student.age}");
    print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Name: John
Age: 20
Roll Number: 1
```

Example 2: Parameterized Constructor With Named Parameters In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```dart
class Student {
  String? name;
  int? age;
  int? rollNumber;

  // Constructor
  Student({String? name, int? age, int? rollNumber}) {
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}

void main(){
    // Here student is object of class Student.
    Student student = Student(name: "John", age: 20, rollNumber: 1);
    print("Name: ${student.name}");
    print("Age: ${student.age}");
    print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Name: John
Age: 20
Roll Number: 1
```

Example 3: Parameterized Constructor With Default Values In Dart

In this example below, there is class **Student** with two properties: **name**, and **age**. The class has parameterized constructor with default values. The constructor is used to initialize the values of the two properties. We also have an object of the class **Student** called **student**.

```dart
class Student {
  String? name;
  int? age;

  // Constructor
  Student({String? name = "John", int? age = 0}) {
    this.name = name;
    this.age = age;
  }
}

void main(){
    // Here student is object of class Student.
    Student student = Student();
    print("Name: ${student.name}");
    print("Age: ${student.age}");
}
```

Show Output

```
Name: John
Age: 0
```

Note: In parameterized constructor, at the time of object creation, you must pass the parameters through the constructor which initialize the variables value, avoiding the null values.

NAMED CONSTRUCTOR IN DART

Named Constructor In Dart

In most programming languages like java, c++, c#, etc., we can create multiple constructors with the same name. But in Dart, this is not possible. Well, there is a way. We can create multiple constructors with the same name using **named constructors**.

Note: Named constructors improves code readability. It is useful when you want to create multiple constructors with the same name.

Example 1: Named Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has two constructors. The first constructor is a default constructor. The second constructor is a named constructor. The named constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```
class Student {
  String? name;
  int? age;
  int? rollNumber;

  // Default Constructor
  Student() {
    print("This is a default constructor");
  }

  // Named Constructor
  Student.namedConstructor(String name, int age, int rollNumber) {
    this.name = name;
    this.age = age;
    this.rollNumber = rollNumber;
  }
}

void main() {
  // Here student is object of class Student.
  Student student = Student.namedConstructor("John", 20, 1);
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
This is a default constructor
Name: John
Age: 20
Roll Number: 1
```

Example 2: Named Constructor In Dart

In this example below, there is class **Mobile** with three properties **name**, **color**, and **price**. The class has one method **display** which prints out the values of the three properties. We also have an object of the class **Mobile** called **mobile**. There is also constructor **Mobile** which takes all the three properties as parameters. Named constructor **Mobile.namedConstructor** is used to create an object of the class **Mobile** with name, color and optional price. The default value of the price is 0. If the price is not passed, then the default value is used.

```dart
class Mobile {
  String? name;
  String? color;
  int? price;

  Mobile(this.name, this.color, this.price);
  // here Mobile() is a named constructor
  Mobile.namedConstructor(this.name, this.color, [this.price = 0]);

  void displayMobileDetails() {
    print("Mobile name: $name.");
    print("Mobile color: $color.");
    print("Mobile price: $price");
  }
}

void main() {
  var mobile1 = Mobile("Samsung", "Black", 20000);
  mobile1.displayMobileDetails();
  var mobile2 = Mobile.namedConstructor("Apple", "White");
  mobile2.displayMobileDetails();
}
```

Show Output

```
Mobile name: Samsung.
Mobile color: Black.
Mobile price: 20000
Mobile name: Apple.
Mobile color: White.
Mobile price: 0
```

Example 3: Named Constructor In Dart

In this example below, there is a class **Animal** with two properties **name** and **age**. The class has three constructors. The first constructor is a default constructor. The second and third constructors are named constructors. The second constructor is used to initialize the values of name and age, and the third constructor is used to initialize the value of name only. We also have an object of the class **Animal** called **animal**.

```dart
class Animal {
  String? name;
  int? age;

  // Default Constructor
  Animal() {
    print("This is a default constructor");
  }

  // Named Constructor
  Animal.namedConstructor(String name, int age) {
    this.name = name;
    this.age = age;
  }

  // Named Constructor
  Animal.namedConstructor2(String name) {
    this.name = name;
  }
}
void main(){
  // Here animal is object of class Animal.
  Animal animal = Animal.namedConstructor("Dog", 5);
  print("Name: ${animal.name}");
  print("Age: ${animal.age}");

  Animal animal2 = Animal.namedConstructor2("Cat");
  print("Name: ${animal2.name}");
}
```

Show Output

```
Name: Dog
Age: 5
Name: Cat
```

Example 4: Real Life Example Of Named Constructor In Dart

In this example below, there is a class **Person** with two properties **name** and **age**. The class has three constructors. The first is a parameterized constructor which takes two parameters **name** and **age**. The second and third constructors are named constructors.

Second constructor fromJson is used to create an object of the class **Person** from a JSON. The third fromJsonString is used to create an object of the class **Person** from a JSON string. We also have an object of the class **Person** called **person**.

```
import 'dart:convert';

class Person {
  String? name;
  int? age;

  Person(this.name, this.age);

  Person.fromJson(Map<String, dynamic> json) {
    name = json['name'];
    age = json['age'];
  }

  Person.fromJsonString(String jsonString) {
    Map<String, dynamic> json = jsonDecode(jsonString);
    name = json['name'];
    age = json['age'];
  }
}

void main() {
// Here person is object of class Person.
  String jsonString1 = '{"name": "Bishworaj", "age": 25}';
  String jsonString2 = '{"name": "John", "age": 30}';

  Person p1 = Person.fromJsonString(jsonString1);
  print("Person 1 name: ${p1.name}");
  print("Person 1 age: ${p1.age}");

  Person p2 = Person.fromJsonString(jsonString2);
  print("Person 2 name: ${p2.name}");
  print("Person 2 age: ${p2.age}");
}
```

Show Output

```
Person 1 name: Bishworaj
Person 1 age: 25
Person 2 name: John
Person 2 age: 30
```

Challenge

Try to create a class **Car** with three properties **name**, **color**, and **price** and one method **display** which prints out the values of the three properties. Create a constructor, which takes all 3 parameters. Create a named constructor which takes two parameters **name** and **color**. Create an object of the class from both the constructors and call the method **display**.

CONSTANT CONSTRUCTOR IN DART

Constant Constructor In Dart

Constant constructor is a constructor that creates a constant object. A constant object is an object whose value cannot be changed. A constant constructor is declared using the keyword **const**.

Note: **Constant Constructor** is used to create a object whose value cannot be changed. It Improves the performance of the program.

Rule For Declaring Constant Constructor In Dart

- All properties of the class must be final.
- It does not have any body.
- Only class containing **const** constructor is initialized using the **const** keyword.

Example 1: Constant Constructor In Dart

In this example below, there is a class **Point** with two final properties: **x** and **y**. The class also has a constant constructor that initializes the two properties. The class also has a method called **display**, which prints out the values of the two properties.

```dart
class Point {
  final int x;
  final int y;

  const Point(this.x, this.y);
}

void main() {
  // p1 and p2 has the same hash code.
  Point p1 = const Point(1, 2);
  print("The p1 hash code is: ${p1.hashCode}");

  Point p2 = const Point(1, 2);
  print("The p2 hash code is: ${p2.hashCode}");
  // without using const
  // this has different hash code.
  Point p3 = Point(2, 2);
  print("The p3 hash code is: ${p3.hashCode}");

  Point p4 = Point(2, 2);
  print("The p4 hash code is: ${p4.hashCode}");
}
```

Show Output

```
The p1 hash code is: 918939239
The p2 hash code is: 918939239
The p3 hash code is: 745146896
The p4 hash code is: 225789186
```

Note: Here p1 and p2 has the same hash code. This is because p1 and p2 are constant objects. The hash code of a constant object is the same. This is because the hash code of a constant object is computed at compile time. The hash code of a non-constant object is computed at run time. This is why p3 and p4 have different hash code.

Example 2: Constant Constructor In Dart

In this example below, there is a class **Student** with three properties: **name**, **age**, and **rollNumber**. The class has one constant constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Student** called **student**.

```
class Student {
  final String? name;
  final int? age;
  final int? rollNumber;

  // Constant Constructor
  const Student({this.name, this.age, this.rollNumber});
}

void main() {
  // Here student is object of Student.
  const Student student = Student(name: "John", age: 20, rollNumber: 1);
  print("Name: ${student.name}");
  print("Age: ${student.age}");
  print("Roll Number: ${student.rollNumber}");
}
```

Show Output

```
Name: John
Age: 20
Roll Number: 1
```

Example 3: Constant Constructor With Named Parameters In Dart

In this example below, there is a class **Car** with three properties: **name**, **model**, and **price**. The class has one constructor. The constructor is used to initialize the values of the three properties. We also have an object of the class **Car** called **car**.

```
class Car {
  final String? name;
  final String? model;
  final int? price;

  // Constant Constructor
  const Car({this.name, this.model, this.price});
}

void main() {
  // Here car is object of class Car.
  const Car car = Car(name: "BMW", model: "X5", price: 50000);
  print("Name: ${car.name}");
  print("Model: ${car.model}");
  print("Price: ${car.price}");
}
```

Show Output

```
Name: BMW
Model: X5
Price: 50000
```

Benefits Of Constant Constructor In Dart

- Improves the performance of the program.

Challenge

Create a class **Customer** with three properties: **name**, **age**, and **phone**. The class should have one constant constructor. The constructor should initialize the values of the three properties. Create an object of the class **Customer** and print the values of the three properties.

ENCAPSULATION IN DART

Introduction

In this section, you will learn about **encapsulation in Dart** programming language with examples. Encapsulation is one of the important concepts of object-oriented programming. Before learning about dart encapsulation, you should have a basic understanding of the **class** and **object** in dart.

Encapsulation In Dart

In Dart, **Encapsulation** means **hiding data** within a library, preventing it from outside factors. It helps you control your program and prevent it from becoming too complicated.

What Is Library In Dart?

By default, every **.dart** file is a library. A library is a collection of functions and classes. A library can be imported into another library using the **import** keyword.

How To Achieve Encapsulation In Dart?

Encapsulation can be achieved by:

- Declaring the class properties as **private** by using **underscore(_)**.
- Providing public **getter** and **setter** methods to access and update the value of private property.

Note: Dart doesn't support keywords like **public**, **private**, and **protected**. Dart uses _ (underscore) to make a property or method private. The encapsulation happens at library level, not at class level.

Getter and Setter Methods

Getter and **setter** methods are used to access and update the value of private property. **Getter** methods are used to access the value of private property. **Setter** methods are used to update the value of private property.

Example 1: Encapsulation In Dart

In this example, we will create a class named **Employee**. The class will have two private properties **_id** and **_name**. We will also create two public methods **getId()** and **getName()** to access the private properties. We will also create two public methods **setId()** and **setName()** to update the private properties.

```
class Employee {
  // Private properties
  int? _id;
  String? _name;

// Getter method to access private property _id
  int getId() {
    return _id!;
  }
// Getter method to access private property _name
  String getName() {
    return _name!;
  }
// Setter method to update private property _id
  void setId(int id) {
    this._id = id;
  }
// Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }

}

void main() {
  // Create an object of Employee class
  Employee emp = new Employee();
  // setting values to the object using setter
  emp.setId(1);
  emp.setName("John");

  // Retrieve the values of the object using getter
  print("Id: ${emp.getId()}");
  print("Name: ${emp.getName()}");
}
```

Show Output

```
Id: 1
Name: John
```

Private Properties

Private property is a property that can only be accessed from same **library**. Dart does not have any keywords like **private** to define a private property. You can define it by prefixing an **underscore (_)** to its name.

Example 2: Private Properties In Dart

In this example, we will create a class named **Employee**. The class has one private property **_name**. We will also create a public method **getName()** to access the private property.

```
class Employee {
  // Private property
  var _name;

  // Getter method to access private property _name
  String getName() {
    return _name;
  }


  // Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }
}

void main() {
  var employee = Employee();
  employee.setName("Jack");
  print(employee.getName());
}
```
Show Output

```
Jack
```

Why Aren’t Private Properties Private?

In the main method, if you write the following code, it will compile and run without any error. Let’s see why it is happening.

```
class Employee {
  // Private property
  var _name;

  // Getter method to access private property _name
  String getName() {
    return _name;
  }

  // Setter method to update private property _name
  void setName(String name) {
    this._name = name;
  }
}

void main() {
  var employee = Employee();
  employee._name = "John"; // It is working, but why?
  print(employee.getName());
}
```
Show Output

```
John
```

Reason

The reason is that using **underscore (_)** before a variable or method name makes it **library private** not **class private**. It means that the variable or method is only visible to the library in which it is declared. It is not visible to any other library. In simple words, library is one file. If you write the main method in a separate file, this will not work.

Solution

To see private properties in action, you must create a separate file for the class and import it into the main file.

Read-only Properties

You can control the properties's access and implement the encapsulation in the dart by using the read-only properties. You can do that by adding the **final** keyword before the properties declaration. Hence, you can only access its value, but you cannot change it.

Note: Properties declared with the **final** keyword must be initialized at the time of declaration. You can also initialize them in the constructor.

```
class Student {
  final _schoolname = "ABC School";

  String getSchoolName() {
    return _schoolname;
  }
}

void main() {
  var student = Student();
  print(student.getSchoolName());
  // This is not possible
  //student._schoolname = "XYZ School";
}
```

Show Output

```
ABC School
```

Note: You can also define **getter** and **setter** using **get** and **set** keywords. For more see this example below.

How To Create Getter and Setter Methods?

You can create getter and setter methods by using the **get** and **set** keywords. In this example below, we have created a class named **Vehicle**. The class has two private properties _**model** and _**year**. We have also created two getter and setter methods for each property. The getter and setter methods are named **model** and **year**. The getter and setter methods are used to access and update the value of the private properties.

```
class Vehicle {
  String _model;
  int _year;

  // Getter method
  String get model => _model;

  // Setter method
  set model(String model) => _model = model;

  // Getter method
  int get year => _year;

  // Setter method
  set year(int year) => _year = year;
}

void main() {
  var vehicle = Vehicle();
  vehicle.model = "Toyota";
  vehicle.year = 2019;
  print(vehicle.model);
  print(vehicle.year);
}
```

Show Output

```
Toyota
2019
```

Note: In dart, any identifier like (class, class properties, top-level function, or variable) that starts with an underscore _ it is private to its library.

Why Encapsulation Is Important?

- **Data Hiding**: Encapsulation hides the data from the outside world. It prevents the data from being accessed by the code outside the class. This is known as data hiding.
- **Testability**: Encapsulation allows you to test the class in isolation. It will enable you to test the class without testing the code outside the class.
- **Flexibility**: Encapsulation allows you to change the implementation of the class without affecting the code outside the class.
- **Security**: Encapsulation allows you to restrict access to the class members. It will enable you to limit access to the class members from the code outside the library.

GETTER IN DART

Getter In Dart

Getter is used to get the value of a property. It is mostly used to access a **private property’s** value. Getter provide explicit read access to an object properties.

Syntax

```
return_type get property_name {
  // Getter body
}
```

Note: Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

Example 1: Getter In Dart

In this example below, there is a class named **Person**. The class has two properties **firstName** and **lastName**. There is getter **fullName** which is responsible to get full name of person.

```
class Person {
  // Properties
  String? firstName;
  String? lastName;

  // Constructor
  Person(this.firstName, this.lastName);

  // Getter
  String get fullName => "$firstName $lastName";
}

void main() {
  Person p = Person("John", "Doe");
  print(p.fullName);
}
```

Show Output

```
John Doe
```

Example 2: Getter In Dart

In this example below, there is a class named **NoteBook**. The class has two private properties _**name** and _**prize**. There are two getters **name** and **price** to access the value of the properties.

```
class NoteBook {
  // Private properties
  String? _name;
  double? _prize;

  // Constructor
  NoteBook(this._name, this._prize);

  // Getter method to access private property _name
  String get name => this._name!;

  // Getter method to access private property _prize
  double get price => this._prize!;
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook("Dell", 500);
  // Display the values of the object
  print(nb.name);
  print(nb.price);
}
```

Show Output

```
Name: Dell
Price: 500.0
```

Note: In the above example, a getter **name** and **price** are used to access the value of the properties _**name** and _**prize**.

Example 3: Getter In Dart With Data Validation

In this example below, there is a class named **NoteBook**. The class has two private properties _**name** and _**prize**. There are two getters **name** and **price** to access the value of the properties. If you provide a blank name, then it will return **No Name**.

```
class NoteBook {
  // Private properties
  String _name;
  double _prize;

  // Constructor
  NoteBook(this._name, this._prize);

  // Getter to access private property _name
  String get name {
    if (_name == "") {
      return "No Name";
    }
    return this._name;
  }

  // Getter to access private property _prize
  double get price {
    return this._prize;
  }
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook("Apple", 1000);
  print("First Notebook name: ${nb.name}");
  print("First Notebook price: ${nb.price}");
  NoteBook nb2 = new NoteBook("", 500);
  print("Second Notebook name: ${nb2.name}");
  print("Second Notebook price: ${nb2.price}");
}
```

Show Output

```
First Notebook name: Apple
First Notebook price: 1000.0
Second Notebook name: No Name
Second Notebook price: 500.0
```

Example 4: Getter In Dart

In this example below, there is a class named **Doctor**. The class has three private properties _**name**, _**age** and _**gender**. There are three getters **name**, **age**, and **gender** to access the value of the properties. It has **map** getter to get Map of the object.

```
class Doctor {
// Private properties
  String _name;
  int _age;
  String _gender;

// Constructor
  Doctor(this._name, this._age, this._gender);

// Getters
  String get name => _name;
  int get age => _age;
  String get gender => _gender;

// Map Getter
```

```
  Map<String, dynamic> get map {
    return {"name": _name, "age": _age, "gender": _gender};
  }
}

void main() {
// Create an object of Doctor class
  Doctor d = Doctor("John", 41, "Male");
  print(d.map);
}
```

Show Output

```
{name: John, age: 41, gender: Male}
```

Why Is Getter Important In Dart?

- To access the value of private property.
- To restrict the access of data members of a class.

Conclusion

In this section, you have learned about **Getter** with the help of examples. In the next section, you will learn about **Setter In Dart**.

SETTER IN DART

Setter In Dart

Setter is used to set the value of a property. It is mostly used to update a **private property's** value. Setter provide explicit write access to an object properties.

Syntax

```
set property_name (value) {
  // Setter body
}
```

Note: Instead of writing { } after the property name, you can also write **=>** (fat arrow) after the property name.

Example 1: Setter In Dart

In this example below, there is a class named **NoteBook**. The class has two private properties _**name** and _**prize**. There are two setters **name** and **price** to update the value of the properties. There is also a method **display** to display the value of the properties.

```
class NoteBook {
  // Private Properties
  String? _name;
  double? _prize;

  // Setter to update private property _name
  set name(String name) => this._name = name;

  // Setter to update private property _prize
  set price(double price) => this._prize = price;

  // Method to display the values of the properties
  void display() {
    print("Name: ${_name}");
    print("Price: ${_prize}");
  }
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook();
  // setting values to the object using setter
  nb.name = "Dell";
  nb.price = 500.00;
  // Display the values of the object
  nb.display();
}
```

Show Output

```
Name: Dell
Price: 500.0
```

Note: In the above example, a setter **name** and **price** are used to update the value of the properties **_name** and **_prize**.

Example 2: Setter In Dart With Data Validation

In this example, there is a class named **NoteBook**. The class has two private properties **_name** and **_prize**. If the value of **_prize** is less than 0, we will throw an exception. There are also two setters **name** and **price** to update the value of the properties. The class also has a method **display()** to display the values of the properties.

```
class NoteBook {
  // Private properties
  String? _name;
  double? _prize;

  // Setter to update the value of name property
  set name(String name) => _name = name;

  // Setter to update the value of price property
  set price(double price) {
    if (price < 0) {
      throw Exception("Price cannot be less than 0");
    }
    this._prize = price;
  }

  // Method to display the values of the properties
  void display() {
    print("Name: $_name");
    print("Price: $_prize");
  }
}

void main() {
  // Create an object of NoteBook class
  NoteBook nb = new NoteBook();
  // setting values to the object using setter
  nb.name = "Dell";
  nb.price = 250;

  // Display the values of the object
  nb.display();
}
```

Show Output

```
Name: Dell
Price: 500.0
```

Note: It is generally best to not allow the user to set the value of a field directly. Instead, you should provide a setter method that can validate the value before setting it. This is very important when working on large and complex programs.

Example 3: Setter In Dart

In this example, there is a class named **Student**. The class has two private properties _**name** and _**classnumber**. We will also create two setters **name** and **classnumber** to update the value of the properties. The **classnumber** setter will only accept a value between 1 and 12. The class also has a method **display()** to display the values of the properties.

```
class Student {
  // Private properties
  String? _name;
  int? _classnumber;

  // Setter to update the value of name property
  set name(String name) => this._name = name;

  // Setter to update the value of classnumber property
  set classnumber(int classnumber) {
    if (classnumber <= 0 || classnumber > 12) {
      throw ('Classnumber must be between 1 and 12');
    }
    this._classnumber = classnumber;
  }

  // Method to display the values of the properties
  void display() {
    print("Name: $_name");
    print("Class Number: $_classnumber");
  }
}
void main() {
  // Create an object of Student class
  Student s = new Student();
  // setting values to the object using setter
  s.name = "John Doe";
  s.classnumber = 12;

  // Display the values of the object
  s.display();

  // This will generate error
  //s.setClassNumber(13);
}
```

Show Output

```
Name: John
Class Number: 10
```

Why Is Setter Important?

- It is used to set the value of a private property.
- It is also used for data validation.
- It gives you better control over the data.

Challenge

Try to create a class named **University** with two private properties **_name** and **_year**. The class will also have two setters **name** and **year** to update the value of the properties. The **year** setter will only accept a value between 1900 and 2023. Also, create a method **display()** to display the values of the properties.

GETTER AND SETTER IN DART

Introduction

In this section, you will learn about **Getter and Setter** in dart with the help of examples.

Getter And Setter

Getter and **Setter** provide explicit read and write access to an object properties. In dart, **get** and **set** are the keywords used to create getter and setter. Getter read the value of property and act as **accessor**. Setter update the value of property and act as **mutator**.

Note: You can use same name for **getter** and **setter**. But, you can't use same name for **getter**, **setter** and **property name**.

Use Of Getter and Setter

- Validate the data before reading or writing.
- Restrict the read and write access to the properties.
- Making the properties read-only or write-only.
- Perform some action before reading or writing the properties.

Example 1: Getter And Setter In Dart

In this example below, there is a class named **Student** with three private properties **_firstName**, **_lastName** and **_age**. There are two getters **fullName** and **age** to get the value of the properties. There are also three setters **firstName**, **lastName** and **age** to update the value of the properties. If **age** is less than 0, it will throw an error.

```
class Student {
  // Private Properties
  String? _firstName;
  String? _lastName;
  int? _age;

  // Getter to get full name
  String get fullName => this._firstName! + " " + this._lastName!;

  // Getter to read private property _age
  int get age => this._age!;

  // Setter to update private property _firstName
```

```dart
  set firstName(String firstName) => this._firstName = firstName;

  // Setter to update private property _lastName
  set lastName(String lastName) => this._lastName = lastName;

  // Setter to update private property _age
  set age(int age) {
    if (age < 0) {
      throw new Exception("Age can't be less than 0");
    }
    this._age = age;
  }
}

void main() {
  // Create an object of Student class
  Student st = new Student();
  // setting values to the object using setter
  st.firstName = "John";
  st.lastName = "Doe";
  st.age = 20;
  // Display the values of the object
  print("Full Name: ${st.fullName}");
  print("Age: ${st.age}");
}
```

Show Output

```
Full Name: John Doe
Age: 20
```

Example 2: Getter And Setter In Dart

In this example below, there is a class named **BankAccount** with one private property _**balance**. There is one getter **balance** to read the value of the property. There are methods **deposit** and **withdraw** to update the value of the _**balance**.

```dart
class BankAccount {
  // Private Property
  double _balance = 0.0;

  // Getter to read private property _balance
  double get balance => this._balance;

  // Method to deposit money
  void deposit(double amount) {
    this._balance += amount;
  }
  // Method to withdraw money
  void withdraw(double amount) {
    if (this._balance >= amount) {
      this._balance -= amount;
    } else {
      throw new Exception("Insufficient Balance");
    }
  }
}

void main() {
  // Create an object of BankAccount class
  BankAccount account = new BankAccount();
  // Deposit money
  account.deposit(1000);
  // Display the balance
  print("Balance after deposit: ${account.balance}");
  // Withdraw money
  account.withdraw(500);
  // Display the balance
  print("Balance after withdraw: ${account.balance}");
}
```

Show Output

```
Balance after deposit: 1000
Balance after withdraw: 500
```

When To Use Getter And Setter

- Use getter and setter when you want to restrict the access to the properties.
- Use getter and setter when you want to perform some action before reading or writing the properties.
- Use getter and setter when you want to validate the data before reading or writing the properties.
- Don't use getter and setter when you want to make the properties read-only or write-only.

INHERITANCE IN DART

Introduction

In this section, you will learn inheritance in Dart programming and how to define a class that reuses the properties and methods of another class.

Inheritance In Dart

Inheritance is a sharing of behaviour between two classes. It allows you to define a class that extends the functionality of another class. The **extend** keyword is used for inheriting from parent class.

Note: Whenever you use inheritance, it always create a **is-a** relation between the parent and child class like **Student is a Person**, **Truck is a Vehicle**, **Cow is a Animal** etc.

Dart supports single inheritance, which means that a class can only inherit from a single class. Dart does not support multiple inheritance which means that a class cannot inherit from multiple classes.

Syntax

```
class ParentClass {
  // Parent class code
}

class ChildClass extends ParentClass {
  // Child class code
}
```

In this syntax, **ParentClass** is the super class and **ChildClass** is the sub class. The **ChildClass** inherits the properties and methods of the **ParentClass**.

Terminology

Parent Class: The class whose properties and methods are inherited by another class is called parent class. It is also known as base class or super class.

Child Class: The class that inherits the properties and methods of another class is called child class. It is also known as derived class or sub class.

Example 1: Inheritance In Dart

In this example, we will create a class **Person** and then create a class **Student** that inherits the properties and methods of the **Person** class.

```
class Person {
  // Properties
  String? name;
  int? age;

  // Method
  void display() {
    print("Name: $name");
    print("Age: $age");
  }
}
// Here In student class, we are extending the
// properties and methods of the Person class
class Student extends Person {
  // Fields
  String? schoolName;
  String? schoolAddress;

  // Method
  void displaySchoolInfo() {
    print("School Name: $schoolName");
    print("School Address: $schoolAddress");
  }
}

void main() {
  // Creating an object of the Student class
  var student = Student();
  student.name = "John";
  student.age = 20;
  student.schoolName = "ABC School";
  student.schoolAddress = "New York";
  student.display();
  student.displaySchoolInfo();
}
```

Show Output

```
Name: John
Age: 20
School Name: ABC School
School Address: New York
```

Advantages Of Inheritance In Dart

- It promotes reusability of the code and reduces redundant code.
- It helps to design a program in a better way.
- It makes code simpler, cleaner and saves time and money on maintenance.
- It facilitates the creation of class libraries.
- It can be used to enforce standard interface to all children classes.

Example 2: Inheritance In Dart

In this example, here is parent class **Car** and child class **Toyota**. The **Toyota** class inherits the properties and methods of the **Car** class.

```
class Car{
  String color;
  int year;

  void start(){
    print("Car started");
  }
}
```

```
class Toyota extends Car{
  String model;
  int price;

  void showDetails(){
    print("Model: $model");
    print("Price: $price");
  }
}

void main(){
  var toyota = Toyota();
  toyota.color = "Red";
  toyota.year = 2020;
  toyota.model = "Camry";
  toyota.price = 20000;
  toyota.start();
  toyota.showDetails();
}
```

Show Output

```
Car started
Model: Camry
Price: 20000
```

Types Of Inheritance In Dart

1. **Single Inheritance** - In this type of inheritance, a class can inherit from only one class. In Dart, we can only extend one class at a time.
2. **Multilevel Inheritance** - In this type of inheritance, a class can inherit from another class and that class can also inherit from another class. In Dart, we can extend a class from another class which is already extended from another class.
3. **Hierarchical Inheritance** - In this type of inheritance, parent class is inherited by multiple subclasses. For example, the **Car** class can be inherited by the **Toyota** class and **Honda** class.
4. **Multiple Inheritance** - In this type of inheritance, a class can inherit from multiple classes. **Dart does not support multiple inheritance.** For e.g. **Class Toyota extends Car, Vehicle {}** is not allowed in Dart.

Example 3: Single Inheritance In Dart

In this example below, there is super class named **Car** with two properties **name** and **price**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties.

```
class Car {
  // Properties
  String? name;
  double? price;
}

class Tesla extends Car {
  // Method to display the values of the properties
  void display() {
    print("Name: ${name}");
    print("Price: ${price}");
  }
}

void main() {
  // Create an object of Tesla class
  Tesla t = new Tesla();
  // setting values to the object
  t.name = "Tesla Model 3";
  t.price = 50000.00;
  // Display the values of the object
  t.display();
}
```

Show Output

```
Name: Tesla Model 3
Price: 50000.0
```

Example 4: Multilevel Inheritance In Dart

In this example below, there is super class named **Car** with two properties **name** and **price**. There is sub class named **Tesla** which inherits the properties of the super class. The sub class has a method **display** to display the values of the properties. There is another sub class named **Model3** which inherits the properties of the sub class **Tesla**. The sub class has a property **color** and a method **display** to display the values of the properties.

```
class Car {
// Properties
String? name;
double? price;
}

class Tesla extends Car {
// Method to display the values of the properties
void display() {
  print("Name: ${name}");
  print("Price: ${price}");
}
}

class Model3 extends Tesla {
// Properties
String? color;

// Method to display the values of the properties
void display() {
  super.display();
  print("Color: ${color}");
}
}

void main() {
// Create an object of Model3 class
Model3 m = new Model3();
// setting values to the object
m.name = "Tesla Model 3";
m.price = 50000.00;
m.color = "Red";
// Display the values of the object
m.display();
}
```

Show Output

```
Name: Tesla Model 3
Price: 50000.0
Color: Red
```

Note: Here super keyword is used to call the method of the parent class.

Example 5: Multilevel Inheritance In Dart

In this example below, there is class named **Person** with two properties **name** and **age**. There is sub class named **Doctor** with properties **listofdegrees** and **hospitalname**. There is another subclass named **Specialist** with property **specialization**. The sub class has a method **display** to display the values of the properties.

```
class Person {
  // Properties
  String? name;
  int? age;
}

class Doctor extends Person {
  // Properties
  List<String>? listofdegrees;
  String? hospitalname;

  // Method to display the values of the properties
  void display() {
    print("Name: ${name}");
    print("Age: ${age}");
    print("List of Degrees: ${listofdegrees}");
    print("Hospital Name: ${hospitalname}");
  }
}

class Specialist extends Doctor {
  // Properties
  String? specialization;

  // Method to display the values of the properties
  void display() {
    super.display();
    print("Specialization: ${specialization}");
  }
}

void main() {
  // Create an object of Specialist class
  Specialist s = new Specialist();
  // setting values to the object
  s.name = "John";
  s.age = 30;
  s.listofdegrees = ["MBBS", "MD"];
  s.hospitalname = "ABC Hospital";
  s.specialization = "Cardiologist";
  // Display the values of the object
  s.display();
}
```

Show Output

```
Name: John
Age: 30
List of Degrees: [MBBS, MD]
Hospital Name: ABC Hospital
Specialization: Cardiologist
```

Example 6: Hierarchical Inheritance In Dart

In this example below, there is class named **Shape** with two properties **diameter1** and **diameter2**. There is sub class named **Rectangle** with method **area** to calculate the area of the rectangle. There is another subclass named **Triangle** with method **area** to calculate the area of the triangle.

```dart
class Shape {
  // Properties
  double? diameter1;
  double? diameter2;
}

class Rectangle extends Shape {
  // Method to calculate the area of the rectangle
  double area() {
    return diameter1! * diameter2!;
  }
}

class Triangle extends Shape {
  // Method to calculate the area of the triangle
  double area() {
    return 0.5 * diameter1! * diameter2!;
  }
}

void main() {
  // Create an object of Rectangle class
  Rectangle r = new Rectangle();
  // setting values to the object
  r.diameter1 = 10.0;
  r.diameter2 = 20.0;
  // Display the area of the rectangle
  print("Area of the rectangle: ${r.area()}");

  // Create an object of Triangle class
  Triangle t = new Triangle();
  // setting values to the object
  t.diameter1 = 10.0;
  t.diameter2 = 20.0;
  // Display the area of the triangle
  print("Area of the triangle: ${t.area()}");
}
```

Show Output

```
Area of the rectangle: 200.0
Area of the triangle: 100.0
```

Key Points

- Inheritance is used to reuse the code.
- Inheritance is a concept which is achieved by using the **extends** keyword.
- Properties and methods of the super class can be accessed by the sub class.
- Class **Dog** extends class **Animal**{} means Dog is sub class and Animal is super class.
- The sub class can have its own properties and methods.

Why Dart Does Not Support Multiple Inheritance?

Dart does not support multiple inheritance because it can lead to ambiguity. For example, if class **Apple** inherits class **Fruit** and class **Vegetable**, then there may be two methods with the same name **eat**. If the method is called, then which method should be called? This is the reason why Dart does not support multiple inheritance.

What's problem Of Copy Paste Instead Of Inheritance?

If you copy the code from one class to another class, then you will have to maintain the code in both the classes. If you make any changes in one class, then you will have to make the same changes in the other class. This can lead to errors and bugs in the code.

Does Inheritance Finished If I Learned Extending Class?

No, there is a lot more to learn about inheritance. You need to learn about **Constructor Inheritance**, **Method Overriding**, **Abstract Class**, **Interface** and **Mixin** etc. You will learn about these concepts in the next chapters.

INHERITANCE OF CONSTRUCTOR IN DART

Introduction

In this section, you will learn about inheritance of constructor in Dart programming language with the help of examples. Before learning about inheritance of constructor in Dart, you should have a basic understanding of the constructor and inheritance in Dart.

What Is Inheritance Of Constructor In Dart?

Inheritance of constructor in Dart is a process of inheriting the constructor of the parent class to the child class. It is a way of reusing the code of the parent class.

Example 1: Inheritance Of Constructor In Dart

In this example below, there is class named **Laptop** with a constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor.

```dart
class Laptop {
  // Constructor
  Laptop() {
    print("Laptop constructor");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook() {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook();
}
```

Show Output

```
Laptop constructor
MacBook constructor
```

Note: The constructor of the parent class is called first and then the constructor of the child class is called.

Example 2: Inheritance Of Constructor With Parameters In Dart

In this example below, there is class named **Laptop** with a constructor with parameters. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with parameters.

```
class Laptop {
  // Constructor
  Laptop(String name, String color) {
    print("Laptop constructor");
    print("Name: $name");
    print("Color: $color");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook(String name, String color) : super(name, color) {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook("MacBook Pro", "Silver");
}
```

Show Output

```
Laptop constructor
Name: MacBook Pro
Color: Silver
```

```
MacBook constructor
```

Example 3: Inheritance Of Constructor

In this example below, there is class named **Person** with properties **name** and **age**. There is another class named **Student** which extends the **Person** class.
The **Student** class has additional property **rollNumber**. Lets see how to create a constructor for the **Student** class.

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

  // Constructor
  Person(this.name, this.age);
}

class Student extends Person {
  int rollNumber;

  // Constructor
  Student(String name, int age, this.rollNumber) : super(name, age);
}

void main() {
  var student = Student("John", 20, 1);
  print("Student name: ${student.name}");
  print("Student age: ${student.age}");
  print("Student roll number: ${student.rollNumber}");

}
```

Show Output

```
Student name: John
Student age: 20
Student roll number: 1
```

Example 4: Inheritance Of Constructor With Named Parameters In Dart

In this example below, there is class named **Laptop** with a constructor with named parameters. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters.

```
class Laptop {
  // Constructor
  Laptop({String name, String color}) {
    print("Laptop constructor");
    print("Name: $name");
    print("Color: $color");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook({String name, String color}) : super(name: name, color: color) {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook(name: "MacBook Pro", color: "Silver");
}
```

Show Output

```
Laptop constructor
Name: MacBook Pro
Color: Silver
MacBook constructor
```

Example 5: Calling Named Constructor Of Parent Class In Dart

In this example below, there is class named **Laptop** with one default constructor and one named constructor. There is another class named **MacBook** which extends the **Laptop** class. The **MacBook** class has its own constructor with named parameters. You can call the named constructor of the parent class using the **super** keyword.

```
class Laptop {
  // Default Constructor
  Laptop() {
    print("Laptop constructor");
  }

  // Named Constructor
  Laptop.named() {
    print("Laptop named constructor");
  }
}

class MacBook extends Laptop {
  // Constructor
  MacBook() : super.named() {
    print("MacBook constructor");
  }
}

void main() {
  var macbook = MacBook();
}
```

Show Output

```
Laptop named constructor
MacBook constructor
```

SUPER IN DART

Introduction

In this section, you will learn about Super in Dart programming language with the help of examples. Before learning about Super in Dart, you should have a basic understanding of the constructor and inheritance in Dart.

What Is Super In Dart?

Super is used to refer to the parent class. It is used to call the parent class's properties and methods.

Example 1: Super In Dart

In this example below, the **show()** method of the **MacBook** class calls the **show()** method of the parent class using the **super** keyword.

```
class Laptop {
  // Method
    void show() {
        print("Laptop show method");
    }
}

class MacBook extends Laptop {
    void show() {
        super.show(); // Calling the show method of the parent class
        print("MacBook show method");
    }
}

void main() {
  // Creating an object of the MacBook class
  MacBook macbook = MacBook();
  macbook.show();
}
```

Show Output

```
Laptop show method
MacBook show method
```

Example 2: Accessing Super Properties In Dart

In this example below, the **display()** method of the **Tesla** class calls the **noOfSeats** property of the parent class using the **super** keyword.

```dart
class Car {
  int noOfSeats = 4;
}

class Tesla extends Car {
  int noOfSeats = 6;

  void display() {
    print("No of seats in Tesla: $noOfSeats");
    print("No of seats in Car: ${super.noOfSeats}");
  }
}

void main() {
  var tesla = Tesla();
  tesla.display();
}
```

Show Output

```
No of seats in Tesla: 6
No of seats in Car: 4
```

Example 3: Super With Constructor In Dart

In this example below, the **Manager** class constructor calls the **Employee** class constructor using the **super** keyword.

```dart
class Employee {
  // Constructor
  Employee(String name, double salary) {
    print("Employee constructor");
    print("Name: $name");
    print("Salary: $salary");
  }
}

class Manager extends Employee {
  // Constructor
  Manager(String name, double salary) : super(name, salary) {
    print("Manager constructor");
  }
}
```

```dart
void main() {
  Manager manager = Manager("John", 25000.0);
}
```

Show Output

```
Employee constructor
Name: John
Salary: 25000.0
Manager constructor
```

Example 4: Super With Named Constructor In Dart

In this example below, the **Manager** class named constructor calls the **Employee** class named constructor using the **super** keyword.

```dart
class Employee {
  // Named constructor
  Employee.manager() {
    print("Employee named constructor");
  }
}

class Manager extends Employee {
  // Named constructor
  Manager.manager() : super.manager() {
    print("Manager named constructor");
  }
}

void main() {
  Manager manager = Manager.manager();
}
```

Show Output

```
Employee named constructor
Manager named constructor
```

Example 5: Super With Multilevel Inheritance In Dart

In this example below, the **MacBookPro** class method **display** calls the **display** method of the parent class **MacBook** using the **super** keyword. The **MacBook** class method **display** calls the **display** method of the parent class **Laptop** using the **super** keyword.

```dart
class Laptop {
  // Method
  void display() {
    print("Laptop display");
  }
}

class MacBook extends Laptop {
  // Method
  void display() {
    print("MacBook display");
    super.display();
  }
}

class MacBookPro extends MacBook {
  // Method
  void display() {
    print("MacBookPro display");
    super.display();
  }
}

void main() {
  var macbookpro = MacBookPro();
  macbookpro.display();
}
```

Show Output

```
MacBookPro display
MacBook display
Laptop display
```

Key Points To Remember

- The **super** keyword is used to access the parent class members.
- The **super** keyword is used to call the method of the parent class.

POLYMORPHISM IN DART

Introduction

In this section, you will learn about polymorphism in Dart programming language with the help of examples. Before learning about polymorphism in Dart, you should have a basic understanding of the inheritance in Dart.

Polymorphism In Dart

Poly means **many** and morph means **forms**. Polymorphism is the ability of an object to take on many forms. As humans, we have the ability to take on many forms. We can be a student, a teacher, a parent, a friend, and so on. Similarly, in object-oriented programming, polymorphism is the ability of an object to take on many forms.

Note: In the real world, polymorphism is updating or modifying the feature, function, or implementation that already exists in the parent class.

Polymorphism By Method Overriding

Method overriding is a technique in which you can create a method in the child class that has the same name as the method in the parent class. The method in the child class overrides the method in the parent class.

Syntax

```
class ParentClass{
void functionName(){
  }
}
class ChildClass extends ParentClass{
@override
void functionName(){
  }
}
```

Example 1: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Animal** with a method named **eat()**. The **eat()** method is overridden in the child class named **Dog**.

```
class Animal {
  void eat() {
    print("Animal is eating");
  }
}

class Dog extends Animal {
  @override
  void eat() {
    print("Dog is eating");
  }
}

void main() {
  Animal animal = Animal();
  animal.eat();

  Dog dog = Dog();
  dog.eat();
}
```

Show Output

```
Animal is eating
```

```
Dog is eating
```

Example 2: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Vehicle** with a method named **run()**. The **run()** method is overridden in the child class named **Bus**.

```
class Vehicle {
  void run() {
    print("Vehicle is running");
  }
}

class Bus extends Vehicle {
  @override
  void run() {
    print("Bus is running");
  }
}

void main() {
  Vehicle vehicle = Vehicle();
  vehicle.run();

  Bus bus = Bus();
  bus.run();
}
```

Show Output

```
Vehicle is running
Bus is running
```

Note: If you don’t write **@override**, the program still runs. But, it is a good practice to write **@override**.

Example 3: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Car** with a method named **power()**. The **power()** method is overridden in two child classes named **Honda** and **Tesla**.

```
class Car{
  void power(){
    print("It runs on petrol.");
  }
}

class Honda extends Car{
}
class Tesla extends Car{
  @override
  void power(){
    print("It runs on electricity.");
  }
}

void main(){
  Honda honda=Honda();
  Tesla tesla=Tesla();

  honda.power();
  tesla.power();
}
```

Show Output

```
It runs on petrol.
It runs on electricity.
```

Example 4: Polymorphism By Method Overriding In Dart

In this example below, there is a class named **Employee** with a method named **salary()**. The **salary()** method is overridden in two child classes named **Manager** and **Developer**.

```
class Employee{
  void salary(){
    print("Employee salary is \$1000.");
  }
}

class Manager extends Employee{
  @override
  void salary(){
    print("Manager salary is \$2000.");
  }
}

class Developer extends Employee{
  @override
  void salary(){
    print("Developer salary is \$3000.");
  }
}

void main(){
  Manager manager=Manager();
  Developer developer=Developer();

  manager.salary();
  developer.salary();
}
```

Show Output

```
Manager salary is $2000.
Developer salary is $3000.
```

Advantage Of Polymorphism In Dart

- Subclasses can override the behavior of the parent class.
- It allows us to write code that is more flexible and reusable.

STATIC IN DART

Introduction

In this section, you will learn about **dart static** to share the same variable or method across all instances of a class.

Static In Dart

If you want to define a variable or method that is shared by all instances of a class, you can use the **static** keyword. Static members are accessed using the class name. It is used for **memory management**.

Dart Static Variable

A static variable is a variable that is shared by all instances of a class. It is declared using the static keyword. It is initialized only once when the class is loaded. It is used to store the **class-level data**.

How To Declare A Static Variable In Dart

To declare a static variable in Dart, you must use the static keyword before the variable name.

```
class ClassName {
  static dataType variableName;
}
```

How To Initialize A Static Variable In Dart

To initialize a static variable simply assign a value to it.

```
class ClassName {
  static dataType variableName = value;
  // for e.g
  // static int num = 10;
  // static String name = "Dart";
}
```

How To Access A Static Variable In Dart

You need to use the **ClassName.variableName** to access a static variable in Dart.

```
class ClassName {
  static dataType variableName = value;
  // Accessing the static variable inside same class
  void display() {
    print(variableName);
  }
}

void main() {
  // Accessing static variable outside the class
  dataType value =ClassName.variableName;
}
```

Example 1: Static Variable In Dart

In this example below, there is a class named **Employee**. The class has a static variable **count** to count the number of employees.

```
class Employee {
  // Static variable
  static int count = 0;
  // Constructor
  Employee() {
    count++;
  }
  // Method to display the value of count
  void totalEmployee() {
    print("Total Employee: $count");
  }
}

void main() {
  // Creating objects of Employee class
  Employee e1 = new Employee();
  e1.totalEmployee();
  Employee e2 = new Employee();
  e2.totalEmployee();
  Employee e3 = new Employee();
  e3.totalEmployee();
}
```
Show Output

```
Total Employee: 1
Total Employee: 2
Total Employee: 3
```

Note: While creating the objects of the class, the static variable **count** is incremented by 1. The **totalEmployee()** method displays the value of the static variable **count**.

Example 2: Static Variable In Dart

In this example below, there is a class named **Student**. The class has a static variable **schoolName** to store the name of the school. If every student belongs to the same school, then it is better to use a static variable.

```
class Student {
  int id;
  String name;
  static String schoolName = "ABC School";
  Student(this.id, this.name);
  void display() {
    print("Id: ${this.id}");
    print("Name: ${this.name}");
    print("School Name: ${Student.schoolName}");
  }
}

void main() {
  Student s1 = new Student(1, "John");
  s1.display();
  Student s2 = new Student(2, "Smith");
  s2.display();
}
```
Show Output

```
Id: 1
Name: John
School Name: ABC School
Id: 2
Name: Smith
School Name: ABC School
```

Dart Static Method

A static method is shared by all instances of a class. It is declared using the static keyword. You can access a static method without creating an object of the class.

Syntax

```
class ClassName{
static returnType methodName(){
  //statements
}
}
```

Example 3: Static Method In Dart

In this example, we will create a static method **calculateInterest()** which calculates the simple interest. You can call **SimpleInterest.calculateInterest()** anytime without creating an instance of the class.

```
class SimpleInterest {
  static double calculateInterest(double principal, double rate, double
time) {
    return (principal * rate * time) / 100;
  }
}

void main() {
  print(
      "The simple interest is ${SimpleInterest.calculateInterest(1000, 2,
2)}");
}
```

Show Output

```
The simple interest is 40.0
```

Example 4: Static Method In Dart

In this example below, there is static method **generateRandomPassword()** which generates a random password. You can call **PasswordGenerator.generateRandomPassword()** anytime without creating an instance of the class.

```
import 'dart:math';

class PasswordGenerator {
  static String generateRandomPassword() {
    List<String> allalphabets = 'abcdefghijklmnopqrstuvwxyz'.split('');
    List<int> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    List<String> specialCharacters = ["@", "#", "%", "&", "*"];
    List<String> password = [];
    for (int i = 0; i < 5; i++) {
      password.add(allalphabets[Random().nextInt(allalphabets.length)]);
      password.add(numbers[Random().nextInt(numbers.length)].toString());
      password
.add(specialCharacters[Random().nextInt(specialCharacters.length)]);
    }
    return password.join();
  }
}

void main() {
  print(PasswordGenerator.generateRandomPassword());
}
```

Show Output

```
v5*p4*o2&c7%k1@
```

Note: You don’t need to create an instance of a class to call a static method.

Key Points To Remember

- Static members are accessed using the class name.
- All instances of a class share static members.

ENUM IN DART

Enum In Dart

An enum is a special type that represents a fixed number of constant values. An enum is declared using the keyword **enum** followed by the enum’s name.

Syntax

```
enum enumName {
  constantName1,
  constantName2,
  constantName3,
  ...
  constantNameN
}
```

Example 1: Enum In Dart

In this example below, there is enum type named **days**. It contains seven constants days. The **days** enum type is used in the **main()** function.

```
enum days {
  Sunday,
  Monday,
  Tuesday,
  Wednesday,
  Thrusday,
  Friday,
  Saturday
}

void main() {
  var today = days.Friday;
  switch (today) {
    case days.Sunday:
      print("Today is Sunday.");
      break;
    case days.Monday:
      print("Today is Monday.");
      break;
    case days.Tuesday:
      print("Today is Tuesday.");
      break;
    case days.Wednesday:
      print("Today is Wednesday.");
      break;
    case days.Thursday:
      print("Today is Thursday.");
      break;
    case days.Friday:
      print("Today is Friday.");
      break;
    case days.Saturday:
      print("Today is Saturday.");
      break;
  }
}
```

Show Output

```
Today is Friday.
```

Example 2: Enum In Dart

In this example, there is an enum type named **Gender**. It contains three constants **Male**, **Female**, and **Other**. The **Gender** enum type is used in the **Person** class.

```
enum Gender { Male, Female, Other }

class Person {
  // Properties
  String? firstName;
  String? lastName;
  Gender? gender;
  // Constructor
  Person(this.firstName, this.lastName, this.gender);

  // display() method
  void display() {
    print("First Name: $firstName");
    print("Last Name: $lastName");
    print("Gender: $gender");
  }
}

void main() {
  Person p1 = Person("John", "Doe", Gender.Male);
  p1.display();

  Person p2 = Person("Menuka", "Sharma", Gender.Female);
  p2.display();
}
```

Show Output

```
First Name: John
Last Name: Doe
Gender: Gender.Male
First Name: Menuka
Last Name: Sharma
Gender: Gender.Female
```

How to Print All Enum Values

In this example, there is enum type named **Days**. It contain 7 days. The for loop iterates through all the enum values.

```
enum Days { Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
}

void main() {
 // Days.values: It returns all the values of the enum.
  for (Days day in Days.values) {
    print(day);
  }
}
```

Show Output

```
Days.Sunday
Days.Monday
Days.Tuesday
Days.Wednesday
Days.Thursday
Days.Friday
Days.Saturday
```

Advantages Of Enum In Dart

- It is used to define a set of named constants.
- Makes your code more readable and maintainable.
- It makes the code more reusable and makes it easier for developers.

Characteristics Of Enum

- It must contain at least one constant value.
- Enums are declared outside the class.
- Used to store a large number of constant values.

Enhanced Enum In Dart

In dart, you can declare enums with members. For example, for your accounting software you can store company types like **Sole Proprietorship**, **Partnership**, **Corporation**, and **Limited Liability Company**. You can declare an enum with members as shown below.

```
enum CompanyType {
  soleProprietorship("Sole Proprietorship"),
  partnership("Partnership"),
  corporation("Corporation"),
  limitedLiabilityCompany("Limited Liability Company");

  // Members
  final String text;
  const CompanyType(this.text);
}

void main() {
  CompanyType soleProprietorship = CompanyType.soleProprietorship;
  print(soleProprietorship.text);
}
```

Show Output

```
Sole Proprietorship
```

ABSTRACT CLASS IN DART

Introduction

In this section, you will learn about **dart abstract class**. Before learning about abstract class, you should have a basic understanding of **class**, **object**, **constructor**, and **inheritance**. Previously you learned how to define a class. These classes are **concrete classes**. You can create an object of concrete classes, but you cannot create an object of abstract classes.

Abstract Class

Abstract classes are classes that cannot be initialized. It is used to define the behavior of a class that can be inherited by other classes. An abstract class is declared using the keyword **abstract**.

Syntax

```
abstract class ClassName {
  //Body of abstract class

  method1();
  method2();
}
```

Abstract Method

An abstract method is a method that is declared without an implementation. It is declared with a semicolon (;) instead of a method body.

Syntax

```
abstract class ClassName {
  //Body of abstract class
  method1();
  method2();
}
```

Why We Need Abstract Class

Subclasses of an abstract class must implement all the abstract methods of the abstract class. It is used to achieve abstraction in the Dart programming language.

Example 1: Abstract Class In Dart

In this example below, there is an abstract class **Vehicle** with two abstract methods **start()** and **stop()**. The subclasses **Car** and **Bike** implement the abstract methods and override them to print the message.

```
abstract class Vehicle {
  // Abstract method
  void start();
  // Abstract method
  void stop();
}

class Car extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Car started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Car stopped');
  }
}

class Bike extends Vehicle {
  // Implementation of start()
  @override
  void start() {
    print('Bike started');
  }

  // Implementation of stop()
  @override
  void stop() {
    print('Bike stopped');
  }
}

void main() {
  Car car = Car();
  car.start();
  car.stop();

  Bike bike = Bike();
  bike.start();
  bike.stop();
}
```

Show Output

```
Car started
Car stopped
Bike started
Bike stopped
```

Note: The abstract class is used to define the behavior of a class that can be inherited by other classes. You can define an abstract method inside an abstract class.

Example 2: Abstract Class In Dart

In this example below, there is an abstract class **Shape** with one abstract method **area()** and two subclasses **Rectangle** and **Triangle**. The subclasses implement the **area()** method and override it to calculate the area of the rectangle and triangle, respectively.

```dart
abstract class Shape {
  int dim1, dim2;
  // Constructor
  Shape(this.dim1, this.dim2);
  // Abstract method
  void area();
}

class Rectangle extends Shape {
  // Constructor
  Rectangle(int dim1, int dim2) : super(dim1, dim2);

  // Implementation of area()
  @override
  void area() {
    print('The area of the rectangle is ${dim1 * dim2}');
  }
}

class Triangle extends Shape {
  // Constructor
  Triangle(int dim1, int dim2) : super(dim1, dim2);

  // Implementation of area()
  @override
  void area() {
    print('The area of the triangle is ${0.5 * dim1 * dim2}');
  }
}

void main() {
  Rectangle rectangle = Rectangle(10, 20);
  rectangle.area();

  Triangle triangle = Triangle(10, 20);
  triangle.area();
}
```

Show Output

```
The area of the rectangle is 200
The area of the triangle is 100.0
```

Constructor In Abstract Class

You can’t create an object of an abstract class. However, you can define a constructor in an abstract class. The constructor of an abstract class is called when an object of a subclass is created.

Example 3: Constructor In Abstract Class

In this example below, there is an abstract class **Bank** with a constructor which takes two parameters **name** and **rate**. There is an abstract method **interest()**. The subclasses **SBI** and **ICICI** implement the abstract method and override it to print the interest rate.

```
abstract class Bank {
  String name;
  double rate;

  // Constructor
  Bank(this.name, this.rate);

  // Abstract method
  void interest();

  //Non-Abstract method: It have an implementation
  void display() {
    print('Bank Name: $name');
  }
}

class SBI extends Bank {
  // Constructor
  SBI(String name, double rate) : super(name, rate);

  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of SBI is $rate');
  }
}

class ICICI extends Bank {
  // Constructor
  ICICI(String name, double rate) : super(name, rate);

  // Implementation of interest()
  @override
  void interest() {
    print('The rate of interest of ICICI is $rate');
  }
}

void main() {
  SBI sbi = SBI('SBI', 8.4);
  ICICI icici = ICICI('ICICI', 7.3);

  sbi.interest();
  icici.interest();
  icici.display();
}
```

Show Output

```
The rate of interest of SBI is 8.4
The rate of interest of ICICI is 7.3
Bank Name: ICICI
```

Key Points To Remember

- You can't create an object of an abstract class.
- It can have both abstract and non-abstract methods.
- It is used to define the behavior of a class that other classes can inherit.
- Abstract method only has a signature and no implementation.

INTERFACE IN DART

Introduction

In this section, you will learn the dart interface and how to implement an interface with the help of examples. In Dart, every class is **implicit interface**. Before learning about the interface in dart, you should have a basic understanding of the class and objects, inheritance and abstract class in Dart.

Interface In Dart

An interface defines a syntax that a class must follow. It is a contract that defines the capabilities of a class. It is used to achieve abstraction in the Dart programming language. When you implement an interface, you must implement all the properties and methods defined in the interface. Keyword **implements** is used to implement an interface.

Syntax Of Interface In Dart

```
class InterfaceName {
  // code
}
```

```
class ClassName implements InterfaceName {
  // code
}
```

Declaring Interface In Dart

In dart there is no keyword **interface** but you can use **class** or **abstract class** to declare an interface. All classes implicitly define an interface. Mostly **abstract class** is used to declare an interface.

```
// creating an interface using abstract class
abstract class Person {
  canWalk();
  canRun();
}
```

Implementing Interface In Dart

You must use the **implements** keyword to implement an interface. The class that implements an interface must implement all the methods and properties of the interface.

```
class Student implements Person {
 // implementation of canWalk()
  @override
  canWalk() {
    print('Student can walk');
  }

// implementation of canRun()
  @override
  canRun() {
    print('Student can run');
  }
}
```

Example 1: Interface In Dart

In this example below, there is an interface **Laptop** with two methods **turnOn()** and **turnOff()**. The class **MacBook** implements the interface and overrides the methods to print the message.

```
// creating an interface using concrete class
class Laptop {
    // method
  turnOn() {
    print('Laptop turned on');
  }
    // method
  turnOff() {
    print('Laptop turned off');
  }
}

class MacBook implements Laptop {
  // implementation of turnOn()
  @override
  turnOn() {
    print('MacBook turned on');
  }

  // implementation of turnOff()
  @override
  turnOff() {
    print('MacBook turned off');
  }
}

void main() {
  var macBook = MacBook();
  macBook.turnOn();
  macBook.turnOff();
}
```

Show Output

```
MacBook turned on
MacBook turned off
```

Note: Most of the time, **abstract class** is used instead of **concrete class** to declare an interface.

Example 2: Interface In Dart

In this example below, there is an abstract class named **Vehicle**. The **Vehicle** class has two abstract methods **start()** and **stop()**. The **Car** class implements the **Vehicle** interface. The **Car** class has to implement the **start()** and **stop()** methods.

```
// abstract class as interface
abstract class Vehicle {
  void start();
  void stop();
}
// implements interface
class Car implements Vehicle {
  @override
  void start() {
    print('Car started');
  }

  @override
  void stop() {
    print('Car stopped');
  }
}

void main() {
  var car = Car();
  car.start();
  car.stop();
}
```
Show Output

```
Car started
Car stopped
```
Multiple Inheritance In Dart

Multiple inheritance means a class can inherit from more than one class. In dart, you can’t inherit from more than one class. But you can implement multiple interfaces in a class.

Syntax For Implementing Multiple Interfaces In Dart
```
class ClassName implements Interface1, Interface2, Interface3 {
  // code
}
```
Example 3: Interface In Dart With Multiple Interfaces

In this example below, two abstract classes are named **Area** and **Perimeter**. The **Area** class has an abstract method **area()** and the **Perimeter** class has an abstract method **perimeter()**. The **Shape** class implements both the **Area** and **Perimeter** classes. The **Shape** class has to implement the **area()** and **perimeter()** methods.

```
// abstract class as interface
abstract class Area {
  void area();
}
// abstract class as interface
abstract class Perimeter {
  void perimeter();
}
// implements multiple interfaces
class Rectangle implements Area, Perimeter {
    // properties
  int length, breadth;

 // constructor
  Rectangle(this.length, this.breadth);

// implementation of area()
  @override
  void area() {
    print('The area of the rectangle is ${length * breadth}');
  }
// implementation of perimeter()
  @override
  void perimeter() {
    print('The perimeter of the rectangle is ${2 * (length + breadth)}');
  }
}

void main() {
  Rectangle rectangle = Rectangle(10, 20);
  rectangle.area();
  rectangle.perimeter();
}
```

Show Output

```
The area of the rectangle is 200
The perimeter of the rectangle is 60
```

Example 4: Interface In Dart

In this example below, there is an abstract class named **Person**. The **Person** class has one property **name** and two abstract methods **run** and **walk**. The **Student** class implements the **Person** interface. The **Student** class has to implement the **run** and **walk** methods.

```
// abstract class as interface
abstract class Person {
    // properties
  String? name;
  // abstract method
  void run();
  void walk();
}

class Student implements Person {
    // properties
  String? name;

 // implementation of run()
 @override
  void run() {
    print('Student is running');
  }
  // implementation of walk()
  @override
  void walk() {
    print('Student is walking');
  }
}

void main() {
  var student = Student();
  student.name = 'John';
  print(student.name);
  student.run();
  student.walk();
}
```

Show Output

```
John
Student is running
Student is walking
```

Example 5: Interface In Dart

In this example below, there is abstract class named **CalculateTotal** and **CalculateAverage**. The **CalculateTotal** class has an abstract method **total()** and the **CalculateAverage** class has an abstract method **average()**. The **Student** class implements both the **CalculateTotal** and **CalculateAverage** classes. The **Student** class has to implement the **total()** and **average()** methods.

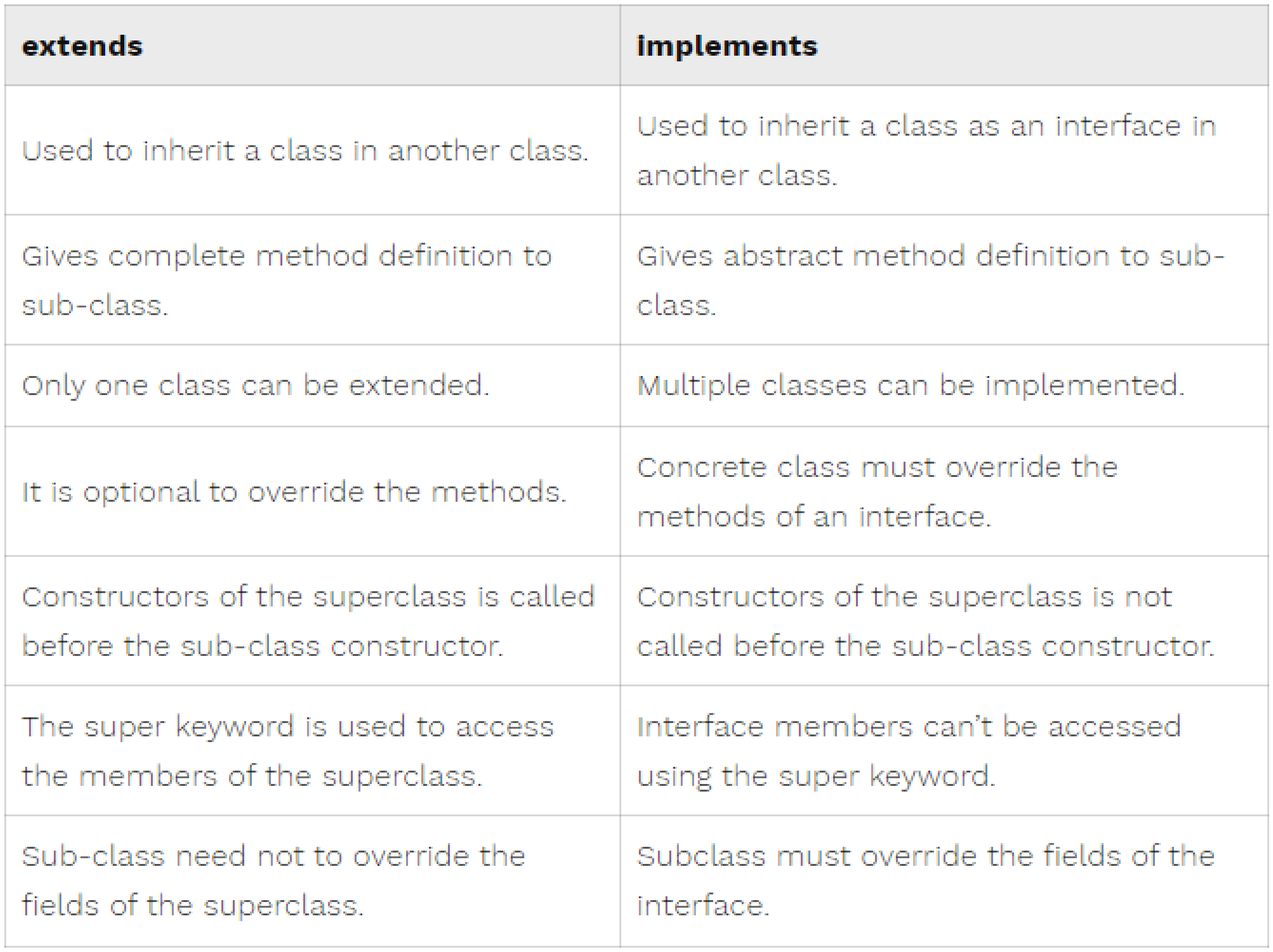
```
// abstract class as interface
abstract class CalculateTotal {
  int total();
}
// abstract class as interface
abstract class CalculateAverage {
  double average();
}
// implements multiple interfaces
class Student implements CalculateTotal, CalculateAverage {
// properties
  int marks1, marks2, marks3;
// constructor
  Student(this.marks1, this.marks2, this.marks3);
// implementation of average()
```

```
  @override
  double average() {
    return total() / 3;
  }
// implementation of total()
  @override
  int total() {
    return marks1 + marks2 + marks3;
  }
}

void main() {
  Student student = Student(90, 80, 70);
  print('Total marks: ${student.total()}');
  print('Average marks: ${student.average()}');
}
```

Show Output

```
Total marks: 240
Average marks: 80.0
```

Difference Between Extends & Implements



extends	implements
Used to inherit a class in another class.	Used to inherit a class as an interface in another class.
Gives complete method definition to sub-class.	Gives abstract method definition to sub-class.
Only one class can be extended.	Multiple classes can be implemented.
It is optional to override the methods.	Concrete class must override the methods of an interface.
Constructors of the superclass is called before the sub-class constructor.	Constructors of the superclass is not called before the sub-class constructor.
The super keyword is used to access the members of the superclass.	Interface members can't be accessed using the super keyword.
Sub-class need not to override the fields of the superclass.	Subclass must override the fields of the interface.

Key Points To Remember

- An interface is a contract that defines the capabilities of a class.
- Dart has no keyword interface, but you can use class or abstract class to declare an interface.
- Use abstract class to declare an interface.
- A class can extend only one class but can implement multiple interfaces.
- Using the interface, you can achieve multiple inheritance in Dart.
- It is used to achieve abstraction.

MIXIN IN DART

Introduction

In this section, you will learn about **dart mixins** to reuse the code in multiple classes. Before learning about mixins you should have a basic understanding of **class**, **object**, **constructor**, and **inheritance**.

Mixin In Dart

Mixins are a way of reusing the code in multiple classes. Mixins are declared using the keyword **mixin** followed by the mixin name. Three keywords are used while working with mixins: **mixin**, **with**, and **on**. It is possible to use multiple mixins in a class.

Note: The **with** keyword is used to apply the mixin to the class. It promotes DRY(Don't Repeat Yourself) principle.

Rules For Mixin

- **Mixin** can't be instantiated. You can't create object of mixin.
- Use the **mixin** to share the code between multiple classes.
- **Mixin** has no constructor and cannot be extended.
- It is possible to use multiple **mixins** in a class.

Syntax

```
mixin Mixin1{
  // code
}

mixin Mixin2{
  // code
}

class ClassName with Mixin1, Mixin2{
  // code
}
```

Example 1: Mixin In Dart

In this example below, there are two mixins named **ElectricVariant** and **PetrolVariant**. The **ElectricVariant** mixin has a method **electricVariant()** and the **PetrolVariant** mixin has a method **petrolVariant()**. The **Car** class uses both the **ElectricVariant** and **PetrolVariant** mixins.

```
mixin ElectricVariant {
  void electricVariant() {
    print('This is an electric variant');
  }
}

mixin PetrolVariant {
  void petrolVariant() {
    print('This is a petrol variant');
  }
}
// with is used to apply the mixin to the class
```

```
class Car with ElectricVariant, PetrolVariant {
  // here we have access of electricVariant() and petrolVariant() methods
}

void main() {
  var car = Car();
  car.electricVariant();
  car.petrolVariant();
}
```

Show Output

```
This is an electric variant
This is a petrol variant
```

Example 2: Mixin In Dart

In this example below, there are two mixins named **CanFly** and **CanWalk**.
The **CanFly** mixin has a method **fly()** and the **CanWalk** mixin has a method **walk()**.
The **Bird** class uses both the **CanFly** and **CanWalk** mixins. The **Human** class uses the **CanWalk** mixin.

```
mixin CanFly {
  void fly() {
    print('I can fly');
  }
}

mixin CanWalk {
  void walk() {
    print('I can walk');
  }
}

class Bird with CanFly, CanWalk {

}

class Human with CanWalk {

}

void main() {
  var bird = Bird();
  bird.fly();
  bird.walk();

  var human = Human();
  human.walk();
}
```
Show Output

```
I can fly
I can walk
I can walk
```

On Keyword

Sometimes, you want to use a mixin only with a specific class. In this case, you can use the **on** keyword.

Syntax Of On Keyword

```
mixin Mixin1 on Class1{
  // code
}
```

Example 3: On Keyword In Mixin In Dart

In this example below, there is abstract class named **Animal** with properties **name** and **speed**. The **Animal** class has an abstract method **run()**. The **CanRun** mixin is only used by class that extends **Animal**. The **Dog** class extends the **Animal** class and uses the **CanRun** mixin. The **Bird** class cannot use the **CanRun** mixin because it does not extend the **Animal** class.

```
abstract class Animal {
  // properties
  String name;
  double speed;

  // constructor
  Animal(this.name, this.speed);

  // abstract method
  void run();
}

// mixin CanRun is only used by class that extends Animal
mixin CanRun on Animal {
  // implementation of abstract method
  @override
  void run() => print('$name is Running at speed $speed');
}

class Dog extends Animal with CanRun {
  // constructor
  Dog(String name, double speed) : super(name, speed);
}
```

```
void main() {
  var dog = Dog('My Dog', 25);
  dog.run();
}

// Not Possible
// class Bird with Animal { }
```

Show Output

```
My Dog is Running at speed 25.0
```

What Is Allowed For Mixin

- You can add properties and static variables.
- You can add regular, abstract, and static methods.
- You can use one or more mixins in a class.

What Is Not Allowed For Mixin

- You can’t define a constructor.
- You can’t extend a mixin.
- You can’t create an object of mixin.

FACTORY CONSTRUCTOR IN DART

Introduction

In this section, you will learn about factory constructors with examples. Before learning about factory constructors, you should have a basic understanding of class and

objects, constructor, abstract class, interface and inheritance in Dart.

Factory Constructor In Dart

All of the constructors that you have learned until now are **generative constructors**. Dart also provides a special type of constructor called a **factory constructor**.

A **factory constructor** gives more flexibility to create an object. Generative constructors only create an instance of the class. But, the factory constructor can return an instance of the **class or even subclass**. It is also used to return the **cached instance** of the class.

Syntax

```
class ClassName {
  factory ClassName() {
    // TODO: return ClassName instance
  }
  factory ClassName.namedConstructor() {
    // TODO: return ClassName instance
  }
}
```

Rules For Factory Constructors

- Factory constructor must return an instance of the **class** or **sub-class**.
- You can’t use **this** keyword inside factory constructor.
- It can be **named** or **unnamed** and called like normal constructor.
- It can’t access **instance members** of the class.

Example 1: Without Factory Constructor

In this example below, there is a class named **Area** with final properties **length** and **breadth**, and **area**. When you pass the **length** and **breadth** to the constructor, it calculates the **area** and stores it in the **area** property.

Note: An initializer list allows you to assign properties to a new instance variable before the constructor body runs, but after creation.

```
class Area {
  final int length;
  final int breadth;
  final int area;

  // Initializer list
 const Area(this.length, this.breadth) : area = length * breadth;
}

void main() {
  Area area = Area(10, 20);
  print("Area is: ${area.area}");

  // notice that here is a negative value
  Area area2 = Area(-10, 20);
  print("Area is: ${area2.area}");
}
```

Show Output

```
Area is: 200
Area is: -200
```

Here **area2** object has a negative value. This is because we are not validating the input. Let’s create a factory constructor to validate the input.

Example 2: With Factory Constructor

In this example below, **factory constructor** is used to validate the input. If the input is valid, it will return a new class instance. If the input is invalid, then it will throw an exception.

```
class Area {
  final int length;
  final int breadth;
  final int area;

  // private constructor
  const Area._internal(this.length, this.breadth) : area = length *
breadth;

  // Factory constructor
  factory Area(int length, int breadth) {
    if (length < 0 || breadth < 0) {
      throw Exception("Length and breadth must be positive");
    }
    // redirect to private constructor
    return Area._internal(length, breadth);
  }
}

void main() {
  // This works
  Area area = Area(10, 20);
  print("Area is: ${area.area}");

  // notice that here is negative value
  Area area2 = Area(-10, 20);
  print("Area is: ${area2.area}");
}
```

Show Output

```
Area is: 200
Unhandled exception:
Exception: Length and breadth must be positive
```

Note: With a factory constructor, you can initialize a final variable using logic that can't be handled in the initializer list.

Example 3: Factory Constructor In Dart

In this example below, there is a class named **Person** with two properties, **firstName** and **lastName**, and two constructors, a **normal constructor** and a **factory constructor**. The factory constructor creates a Person object from a **Map**.

```
class Person {
  String firstName;
  String lastName;

  // constructor
  Person(this.firstName, this.lastName);

  // factory constructor Person.fromMap
  factory Person.fromMap(Map<String, Object> map) {
    final firstName = map['firstName'] as String;
    final lastName = map['lastName'] as String;
    return Person(firstName, lastName);
  }
}

void main() {
  // create a person object
  final person = Person('John', 'Doe');

  // create a person object from map
  final person2 = Person.fromMap({'firstName': 'Harry', 'lastName':
'Potter'});

  // print first and last name
  print("From normal constructor: ${person.firstName}
${person.lastName}");
  print("From factory constructor: ${person2.firstName}
${person2.lastName}");
}
```

In the main method, two objects are created, one using the **generative/normal constructor** and the other using the **factory constructor**.

Show Output

```
From normal constructor: John Doe
From factory constructor: Harry Potter
```

Example 4: Factory Constructor In Dart

In this example below, there is **enum** **ShapeType** with two values: **circle** and **rectangle**. There is an **interface** **Shape** with a factory constructor that creates objects of type Shape, either Circle or Rectangle. The **main** method instantiates two objects, one of each type, and calls the **draw()** method on each.

```
// enum ShapeType
enum ShapeType { circle, rectangle }

// abstract class Shape
abstract class Shape {
  // factory constructor
  factory Shape(ShapeType type) {
    switch (type) {
      case ShapeType.circle:
        return Circle();
      case ShapeType.rectangle:
        return Rectangle();
      default:
        throw 'Invalid shape type';
    }
  }
  // method
  void draw();
}

class Circle implements Shape {
  // implement draw method
  @override
  void draw() {
    print('Drawing circle');
  }
}

class Rectangle implements Shape {
  // implement draw method
  @override
  void draw() {
    print('Drawing rectangle');
  }
}

void main() {
  // create Shape object
  Shape shape = Shape(ShapeType.circle);
  Shape shape2 = Shape(ShapeType.rectangle);
  shape.draw();
  shape2.draw();
}
```
Show Output

```
Drawing circle
Drawing rectangle
```

Note: Here it is possible to make **List** which contains both **Circle** and **Rectangle** objects in it.

Example 5: Factory Constructor In Dart

In this example below, there is class **Person** with a final field **name**. It also has a private constructor and a static _**cache** field. The class also has a **factory constructor** that checks if the _**cache** field contains a key that matches the name parameter. If it does, it returns the Person object associated with that key. Otherwise, it creates a new **Person** object, adds it to the _**cache**, and returns it.

```
class Person {
  // final fields
  final String name;

  // private constructor
  Person._internal(this.name);

  // static _cache field
  static final Map<String, Person> _cache = <String, Person>{};

  // factory constructor
  factory Person(String name) {
    if (_cache.containsKey(name)) {
      return _cache[name]!;
    } else {
      final person = Person._internal(name);
      _cache[name] = person;
      return person;
    }
  }
}

void main() {
  final person1 = Person('John');
  final person2 = Person('Harry');
  final person3 = Person('John');

  // hashcode of person1 and person3 are same
  print("Person1 name is : ${person1.name} with hashcode
${person1.hashCode}");
  print("Person2 name is : ${person2.name} with hashcode
${person2.hashCode}");
```

```
  print("Person3 name is : ${person3.name} with hashcode
${person3.hashCode}");
}
```

Show Output

```
Person1 name is : John with hashcode 117
Person2 name is : Harry with hashcode 118
Person3 name is : John with hashcode 117
```

Singleton In Dart

Singletons are a common design pattern in object-oriented programming. A singleton class can have only one instance and provides a global point of access to it. You can create a singleton in Dart by defining a **factory constructor** that always returns the same instance. It is mostly useful when you want to create a single instance of a class and use it throughout the application like **database connection app**.

Example 6: Singleton Using Factory Constructor

This code creates a **Singleton** class that can only be instantiated once, and provides a factory constructor to get the instance of the class. The main method creates two objects of the Singleton class, and prints the hashcode of the objects to verify that **they are same**.

```
// Singleton using dart factory
class Singleton {
 // static variable
 static final Singleton _instance = Singleton._internal();

// factory constructor
 factory Singleton() {
   return _instance;
 }
 // private constructor
 Singleton._internal();
}

void main() {
 Singleton obj1 = Singleton();
 Singleton obj2 = Singleton();
 print(obj1.hashCode);
 print(obj2.hashCode);
}
```

Show Output

```
2147483648
2147483648
```

You can see that both objects have the same hashcode. This is because both objects are pointing to the same instance.

Note: Here Singleton._internal() is a private constructor so that it can not be called from outside the library. The factory constructor is used to return the same instance of the class.

Key Points

Here **It** means **factory constructor**

- It uses the **factory** keyword to define a factory constructor.
- It returns an instance of the same class or sub-class.
- It is used to implement factory design patterns. [Return sub-class instance based on input parameter as shown in example 4]
- It is used to implement singleton design patterns. [Return the same instance every time]
- It is used to initialize a final variable using logic that can't be handled in the initializer list.

Question For Practice

Create an interface called **Bottle** and add a method to it called **open()**. Create a class called **CokeBottle** and implement the Bottle and print the message **"Coke bottle is opened"**. Add a factory constructor to **Bottle** and return the object of **CokeBottle**. Instantiate **CokeBottle** using the factory constructor and call the **open()** on the object.

GENERIC IN DART

Introduction

This tutorial will teach you about dart Generics, how to create generics classes and methods with examples.

Generics In Dart

Generics is a way to create a class, or function that can work with different types of data **(objects)**. If you look at the internal implementation of **List** class, it is a generic class. It can work with different data types like int, String, double, etc. For example, **List<int>** is a list of integers, **List<String>** is a list of strings, and **List<double>** is a list of double values.

Syntax

```
class ClassName<T> {
  // code
}
```

Example 1: Without Using Generics

Suppose, you need to create a class that can work with both **int** and **double** data types. You can create two classes, one for **int** and another for **double** like this:

```
// Without Generics
// Creating a class for int
class IntData {
  int data;
  IntData(this.data);
}
// Creating a class for double
class DoubleData {
  double data;
  DoubleData(this.data);
}

void main() {
  // Create an object of IntData class
  IntData intData = IntData(10);
  DoubleData doubleData = DoubleData(10.5);
  // Print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

This is not a good practice because both class contain same code. You can create one **Generics** class that can work with different data types. See the example below.

Example 2: Using Generics

In this example below, there is single class that can work with **int**, **double**, and any other data types using **Generics**.

```
// Using Generics
class Data<T> {
  T data;
  Data(this.data);
}

void main() {
  // create an object of type int and double
  Data<int> intData = Data<int>(10);
  Data<double> doubleData = Data<double>(10.5);

  // print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

Generics Type Variable

Generics type variables are used to define the type of data that can be used with the class. In the above example, **T** is a type variable. You can use any name for the type variable. A few typical names are **T**, **E**, **K**, and **V**.

Name	**Work**
T	Type
E	Element
K	Key
V	Value

Dart Map Class

Like **List**, internal implementation of **Map** work with different types of data like int, String, double, etc. This is because Map is a generic class.

```
// Dart implementation of Map class
abstract class Map<K, V> {
  // code
  external factory Map();
}
```

This simply means that the Map class can work with different types of data.

```
void main() {
  final info = {
    "name": "John",
    "age": 20,
    "height": 5.5,
  }
}
```

Generics Methods

You can also create a generic method. For this, you need to use the **<T>** keyword before the method's return type. See the example below.

```
// Define generic method
T genericMethod<T>(T value) {
  return value;
}

void main() {
  // call the generic method
  print("Int: ${genericMethod<int>(10)}");
  print("Double: ${genericMethod<double>(10.5)}");
  print("String: ${genericMethod<String>("Hello")}");
}
```

Show Output

```
Int: 10
Double: 10.5
String: Hello
```

Example 3: Generic Method With Multiple Parameters

In this example below, you will learn to create a generic method with multiple parameters.

```
// Define generic method
T genericMethod<T, U>(T value1, U value2) {
  return value1;
}

void main() {
  // call the generic method
  print(genericMethod<int, String>(10, "Hello"));
  print(genericMethod<String, int>("Hello", 10));
}
```

Show Output

```
10
```

```
Hello
```

Restricting the Type of Data

While implementing generics, you can restrict the type of data that can be used with the class or method. This is done by using the **extends** keyword. See the example below.

Example 4: Generic Class With Restriction

In this example below, there is a **Data** class that works only with **int** and **double** types. It will not work with other types..

```
// Define generic class with bounded type
class Data<T extends num> {
  T data;
  Data(this.data);
}

void main() {
  // create an object of type int and double
  Data<int> intData = Data<int>(10);
  Data<double> doubleData = Data<double>(10.5);
  // print the data
  print("IntData: ${intData.data}");
  print("DoubleData: ${doubleData.data}");
  // Not Possible
  // Data<String> stringData = Data<String>("Hello");
}
```

Show Output

```
IntData: 10
DoubleData: 10.5
```

Example 5: Generic Method With Restriction

In this example below, a generic method **getAverage** takes two parameters of Type **T**, which is considered a **num**. The method returns the average of the two parameters.

```
// Define generic method
double getAverage<T extends num>(T value1, T value2) {
  return (value1 + value2) / 2;
}

void main() {
  // call the generic method
  print("Average of int: ${getAverage<int>(10, 20)}");
  print("Average of double: ${getAverage<double>(10.5, 20.5)}");
}
```

Show Output

```
Average of int: 15
Average of double: 15.5
```

Example 6: Generic Class In Dart

In this example below, there is an abstract class **Shape** with one abstract method called area which returns a double. Also there are two classes that implement Shape, **Circle** and **Rectangle**. There is class **Region** which takes a list of Shape objects and has a method called totalArea which returns the sum of the areas of all the shapes in the list.

```
// abstract class Shape
abstract class Shape {
  // abstract method area
  double get area;
}

// class Circle which implements Shape
class Circle implements Shape {
  // field radius
  final double radius;
  // constructor
  Circle(this.radius);

  // implementation of area method
  @override
  double get area => 3.14 * radius * radius;
}
// class Rectangle which implements Shape
class Rectangle implements Shape {
  // fields width and height
  final double width;
  final double height;
  // constructor
  Rectangle(this.width, this.height);

  // implementation of area method
  @override
  double get area => width * height;
}

// Generic class Region
class Region<T extends Shape> {
  // field shapes
  List<T> shapes;
  // constructor
```

```dart
  Region({required this.shapes});

  // method totalArea
  double get totalArea {
    double total = 0;
    shapes.forEach((shape) {
      total += shape.area;
    });
    return total;
  }
}

void main() {
  // create objects of Circle and Rectangle
  var circle = Circle(10);
  var rectangle = Rectangle(10, 20);
  // create a list of Shape objects
  var region = Region(shapes: [circle, rectangle]);
  // print the total area
  print("Total Area of Region: ${region.totalArea}");
}
```

Show Output

```
Total Area of Region: 514
```

Advantages of Generics

- It solve the problem of type safety.
- It helps to reuse our code.

QUESTIONS FOR PRACTICE 6

Questions For Practice 6

1. Write a dart program to create a class Laptop with properties [id, name, ram] and create 3 objects of it and print all details.
2. Write a dart program to create a class House with properties [id, name, price]. Create a constructor of it and create 3 objects of it. Add them to the list and print all details.
3. Write a dart program to create an enum class for gender [male, female, others] and print all values.
4. Write a dart program to create a class Animal with properties [id, name, color]. Create another class called Cat and extends it from Animal. Add new properties sound in String. Create an object of a Cat and print all details.
5. Write a dart program to create a class Camera with private properties [id, brand, color, price]. Create a getter and setter to get and set values. Also, create 3 objects of it and print all details.
6. Create an interface called **Bottle** and add a method to it called **open()**. Create a class called **CokeBottle** and implement the Bottle and print the message **“Coke bottle is opened”**. Add a factory constructor to **Bottle** and return the object of **CokeBottle**. Instantiate **CokeBottle** using the factory constructor and call the **open()** on the object.

7. Create a simple quiz application using oop that allows users to play and view their score.

Dart Null Safety

In this section, you will learn about the null safety in dart programming language. Here you will learn the following topics:

- Null Safety in Dart,
- Type Promotion in Dart,
- Late Keyword in Dart, and
- Null Safety Exercise.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

NULL SAFETY IN DART

Null Safety

Null safety is a feature in the Dart programming language that helps developers to avoid null errors. This feature is called **Sound Null Safety** in dart. This allows developers to catch null errors at edit time.

Advantage Of Null Safety

- Write safe code.
- Reduce the chances of application crashes.
- Easy to find and fix bugs in code.

Note: Null safety avoids null errors, runtime bugs, vulnerabilities, and system crashes which are difficult to find and fix.

Example 1: Using Null In Variables

In the example below, the variable **age** is a **int** type. If you pass a null value to this variable, it will give an error instantly.

```
void main() {
   int age = null; // give error
}
```

Show Output

```
Error: Compilation failed.
```

Problem With Null

Programmers do have a lot of difficulties while handling null values. They forget that there are **null** values, so the program breaks. In real world **null** mostly acts as **time bomb** for programmers, which is ready to break the program.

Note: Common cause of errors in programming generally comes from not correctly handling null values.

Non-Nullable By Default

In Dart, variables and fields are non-nullable by default, which means that they cannot have a value **null** unless you explicitly allow it.

```
int productid = 20; // non-nullable
int productid = null; // give error
```

How To Declare Null Value

With dart **sound null Safety**, you cannot provide a null value by **default**. If you are 100% sure to use it, then you can use **?** operator after the type declaration.

```
// Declaring a nullable variable by using ?
String? name;
```

This declares a variable **name**, which can be null or a string.

How To Assign Values To Nullable Variables

You can assign a value to nullable variables just like any other variable. However, you can also assign null to them.

```
void main(){
// Declaring a nullable variable by using ?
String? name;
// Assigning John to name
name = "John";
// Assigning null to name
name = null;
}
```

Show Output

How To Use Nullable Variables

You can use nullable variables in many ways. Some of them are shown below:

- You can use **if** statement to check whether the variable is null or not.

- You can use **!** operator, which returns null if the variable is null.
- You can use **??** operator to assign a default value if the variable is null.

```
void main(){
// Declaring a nullable variable by using ?
String? name;
// Assigning John to name
name = "John";
// Assigning null to name
name = null;
// Checking if name is null using if statement
if(name == null){
print("Name is null");
}
// Using ?? operator to assign a default value
String name1 = name ?? "Stranger";
print(name1);
// Using ! operator to return null if name is null
String name2 = name!;
print(name2);
}
```

Show Output

```
Name is null
Stranger
Uncaught TypeError: Cannot read properties of null (reading
'toString')Error: TypeError: Cannot read properties of null (reading
'toString')
```

Example 2: Define List Of Nullable Items

You can also store null in list values. In this example, the **items** is a list of nullable integers. It can contain null values as well as integers.

```
void main() {
  // list of nullable ints
  List<int?> items = [1, 2, null, 4];
  print(items);
}
```

Show Output

```
[1, 2, null, 4]
```

Example 3: Null Safety In Dart Functions

In this example, the function **printAddress** has a parameter **address** which is a **String** type. If you pass a **null** value to this function, it will give a edit-time error.

```
void printAddress(String address) {
  print(address);
}

void main() {
  printAddress(null); // give error
}
```

Show Output

```
Error: Compilation failed.
```

Example 4: Define Function With Nullable Parameter

If you are 100% sure, then you can use **?** for the type declaration. In this example, the function **printAddress** has a parameter **address**, which is a **String?** type. You can pass both null and string values to this function.

```
// address is a nullable string
void printAddress(String? address) {
  print(address);
}
void main() {
  // Passing null to printAddress
  printAddress(null); // Works
}
```

Show Output

```
null
```

Example 5: Null Safety In Dart Class

In the example, the class **Person** has a parameter **name**, which is a **String** type. If you pass a null value to this class, it will give a compile-time error.

```
class Person {
  String name;
  Person(this.name);
}

void main() {
  Person person = Person(null); // give error
}
```

Show Output

```
Error: Compilation failed.
```

Example 6: Define Null To Class Property

In this example, the class **Person** has a parameter **name**, which is a **String?** type. You can pass both null and string values to this class. To define a nullable property in a class, you can use the **?** operator after the type.

```
class Person {
  String? name;
  Person(this.name);
}

void main() {
  Person person = Person(null); // Works
}
```

Show Output

```
null
```

Example 7: Working With Nullable Class Properties

In the example below, the **Profile** class has two nullable properties: **name** and **bio**. The **printProfile** method prints the name and bio of the profile. If the name or bio is **null**, it prints a default value instead.

```
class Profile {
  String? name;
  String? bio;

  Profile(this.name, this.bio);

  void printProfile() {
    print("Name: ${name ?? "Unknown"}");
    print("Bio: ${bio ?? "None provided"}");
  }
}

void main() {
  // Create a profile with a name and bio
  Profile profile1 = Profile("John", "Software engineer and avid reader");
  profile1.printProfile();

  // Create a profile with only a name
  Profile profile2 = Profile("Jane", null);
  profile2.printProfile();

  // Create a profile with only a bio
  Profile profile3 = Profile(null, "Loves to travel and try new foods");
  profile3.printProfile();
  // Create a profile with no name or bio
  Profile profile4 = Profile(null, null);
  profile4.printProfile();
}
```

Show Output

```
Name: John
Bio: Software engineer and avid reader
Name: Jane
Bio: None provided
Name: Unknown
Bio: Loves to travel and try new foods
Name: Unknown
Bio: None provided
```

Important Point In Dart Null Safety

- Null means no value.
- Common error in programming is caused due to null.
- Dart 2.12 introduced **sound null Safety** to solve null problems.
- Non-nullable type is confirmed never to be **null**.

Note: Sometimes you heard word like **NNBD**. It is **Non-Nullable By Default**, which means you can't assign null to a variable by default.

TYPE PROMOTION IN DART

Type Promotion In Dart

Type promotion in dart means that dart automatically converts a value of one type to another type. Dart does this when it knows that the value is of a specific type.

How Type Promotion Works In Dart?

Types Promotion in Dart works in the following ways:

- Promoting from **general types** to **specific subtypes**.
- Promoting from **nullable types** to **non-nullable types**.

Example 1: Promoting From General Types To Specific Subtypes

In this example, the variable **name** is declared as an **Object**. The **Object** class doesn't have a **.length** property. Variable **name** gets promoted from **Object** to **String** so that you can access the **.length** property of the String class.

```
void main(){
Object name = "Pratik";
// print(name.length) will not work because Dart doesn't know that name is
a String

if(name is String) {
// name promoted from Object to String
  print("The length of name is ${name.length}");
}
}
```

Show Output

```
The length of name is 6
```

Example 2: Type Promotion In Dart

In this example, the variable **result** is declared as a **String**. In both **if** and **else** blocks, the variable **result** is assigned a value of type **String**. Therefore, the variable **result** is automatically promoted to a non-nullable type **String**.

```
void main(){
// result is a String
String result;
// result is promoted to a non-nullable type String
if(DateTime.now().hour < 12) {
  result = "Good Morning";
} else {
  result = "Good Afternoon";
}
// display the result
print("Result is $result");
print("Length of result is ${result.length}");
}
```

Show Output

```
Result is Good Afternoon
Length of result is 15
```

Example 3: Type Promotion With Nullable To Non-Nullable Type

In Dart, you can also throw an exception if the variable is null. In this example, method **printLength**, takes a **String** type parameter. If the parameter is null, then it will throw an exception.

```
// method to print the length of the text
void printLength(String? text){
    if(text == null) {
        throw Exception("The text is null");
    }
    print("Length of text is ${text.length}");
}
// main method
void main() {
    printLength("Hello");
}
```

Show Output

```
Length of text is 5
```

Example 4: Type Promotion With Nullable Type To Non-Nullable Type

In this example, the variable **value** contains a value of type **String** or **null**. The variable **value** is promoted to a non-nullable type **String** in the **if** block. If the variable **value** is null, then the **else** block is executed.

```
// importing dart:math library
import 'dart:math';
// creating a class DataProvider
class DataProvider{
    // creating a method stringorNull
    String? get stringorNull => Random().nextBool() ? "Hello" : null;

    // creating a method myMethod
    void myMethod(){
        String? value = stringorNull;
        // checking if value String or not
        if(value is String){
            print("The length of value is ${value.length}");
        }else{
            print("The value is not string.");
        }

    }
}
// main method
void main() {
    DataProvider().myMethod();
}
```

Show Output

```
The length of value is 5
```

Note: The output of the above example is random. It can be either **The length of value is 5** or **The value is not string.**

LATE KEYWORD IN DART

Late Keyword In Dart

In dart, **late** keyword is used to declare a variable or field that will be initialized at a later time. It is used to declare a **non-nullable** variable that is not initialized at the time of declaration.

Example 1: Late Keyword In Dart

In this example, **name** variable is declared as a **late** variable. The **name** variable is initialized in the **main** method.

```
// late variable
late String name;

void main() {
  // assigning value to late variable
  name = "John";
  print(name);
}
```

Show Output

```
John
```

When you put **late** infront of a variable declearation, you tell Dart the following:

- Don't assign that variable a value yet.
- You will assign value later.
- You will make sure the variable has a value before you use it.

Note: The **late** keyword is contract between you and Dart. You are telling Dart that you will assign a value to the variable before you use it. If you don't assign a value to the variable before you use it, Dart will throw an error.

Example 2: Late Keyword In Dart

In this example, there is **Person** class with a **name** field. The **name** field is declared as a late variable.

```
class Person {
  // late variable
  late String name;

  void greet() {
    print("Hello $name");
  }
}

void main() {
  Person person = Person();
  // late variable is initialized here
  person.name = "John";
  person.greet();
}
```

Show Output

```
Hello John
```

Usecase of Late Keyword In Dart

Dart late keyword has two use cases:

- **Declaring a non-nullable variable or field** that is not initialized at the point of declaration.
- **Lazy initialization** of a variable or field.

What Is Lazy Initialization

Lazy initialization is a design pattern that delays the creation of an object, the calculation of a value, or some other expensive process until the **first time you need it**.

Note: Using **late** means dart doesn't initialize value right away, it only initializes when you access it for the first time. This is also called **lazy loading**.

Example 3: Late Keyword In Dart

In this example, the **provideCountry** function is not called when the **value** variable is declared. The **provideCountry** function is called only when the **value** variable is used. **Lazy initialization** is used to avoid unnecessary computation.

```
// function
String provideCountry() {
  print("Function is called");
  return "USA";
}

void main() {
  print("Starting");
  // late variable
  late String value = provideCountry();
  print("End");
  print(value);
}
```

Guess the output before clicking on the **Show Output** button. If you remove the **late** keyword from the **value** variable, the **provideCountry** function will be called when the **value** variable is declared.

Show Output

```
Starting
End
Function is called
USA
```

Example 4: Late Keyword In Class

In this example, the **heavyComputation** function is called when the **description** variable is used. If you remove the **late** keyword from the **description** variable, the **heavyComputation** function will be called when the **Person** class is instantiated.

```
// Person class
class Person {
  final int age;
  final String name;
  late String description = heavyComputation();

// constructor
  Person(this.age, this.name) {
    print("Constructor is called");
  }
// method
  String heavyComputation() {
    print("heavyComputation is called");
    return "Heavy Computation";
  }
}

void main() {
  // object of Person class
  Person person = Person(10, "John");
  print(person.name);
  print(person.description);
}
```

Show Output

```
Constructor is called
John
heavyComputation is called
```

```
Heavy Computation
```

Example 5: Late Keyword In Class

In this example, the **_getFullName** function is called when the **fullName** variable is used. The **firstName** and **lastName** variables are initialized when the **fullName** variable is used.

```dart
class Person {
  // declaring late variables
  late String fullName = _getFullName();
  late String firstName = fullName.split(" ").first;
  late String lastName = fullName.split(" ").last;

// method
  String _getFullName() {
    print("_getFullName is called");
    return "John Doe";
  }
}
// main method
void main() {
  print("Start");
  Person person = Person();
  print("First Name: ${person.firstName}");
  print("Last Name: ${person.lastName}");
  print("Full Name: ${person.fullName}");
  print("End");
}
```

Show Output

```
Start
_getFullName is called
First Name: John
Last Name: Doe
Full Name: John Doe
End
```

Note: If you remove the **late** keyword from the **fullName** variable, the **_getFullName** function will be called when the **Person** class is instantiated.

Late Final Keyword In Dart

If you want to assign a value to a variable only once, you can use the **late final** keyword. This is useful when you want to initialize a variable only once.

Example 6: Late Final Keyword In Dart

In this example, there is class **Student** with a **name** field. The **name** field is declared as a **late final** variable. The **name** field is initialized in the **Student** constructor. The **name** field is assigned a value only once. If you try to assign a value to the **name** field again, you will get an error.

```
// Student class
class Student {
  // late final variable
  late final String name;

  // constructor
  Student(this.name);
}

void main() {
  // object of Student class
  Student student = Student("John");
  print(student.name);
  student.name = "Doe"; // Error
}
```

Show Output

```
John
Unhandled exception:
LateInitializationError: Field 'name' has already been initialized.
```

NULL SAFETY EXERCISE

Null Safety Exercise

Practice these exercises to master **dart null safety**. To practice these exercises, click on **Run Online** button and solve the problem.

Exercise 1: Null Safety In Dart

In variable name **age**, assign a **null** value to it using **?**.

```
// Try to assign a null value to age variable using ?
void main() {
  int age;
  age = null;
  print("Age is $age");
}
```

Exercise 2: Nullable Type Parameter For Generics

Try using **?** to make the type parameter of **List** nullable.

```
// Try to make the type parameter of List nullable
void main() {
  List<int> items = [1, 2, null, 4];
  print(items);
}
```

Exercise 3: Null Assertion Operator (!)

Try using null assertion operator **!** to print null if the variable is null.

```
// Try to use null assertion operator(!) to print null if the variable is
null
void main() {
  String? name;
  name = null;
  String name1 = name;
  print(name1);
}
```

Exercise 4: Null Assertion Operator (!) For Generics

Try using null assertion operator **!** to print null if the variable is null.

```
// Try to use null assertion operator(!) to print null if the variable is
null
void main() {
  List<int?> items = [1, 2, null, 4];

  int firstItem = items.first;

  print(firstItem);
}
```

Exercise 5: Null Assertion Operator (!) For Generics

Try using null assertion operator **!** to print null if the variable is null.

```
// Try to use null assertion operator(!) to print null if the variable is
null
int? returnNullButSometimesNot() {
  return -5;
}

void main() {
 int result = returnNullButSometimesNot().abs();
 print(result);
}
```

**Exercise 6: Null Assertion Operator (!) **

Try using null assertion operator **!** to print the length of the String or return null if the variable is null.

```
// Try to use null assertion operator(!) to print the length of the String
or return null if the variable is null
int findLength(String? name) {
    // add null assertion operator here
  return name.length;
}

void main() {
  int? length = findLength("Hello");
  print("The length of the string is $length");
}
```

Exercise 7: Null Coalescing Operator (??)

If you want to assign a default value to a variable if it is null, you can use null coalescing operator **??**.

Try using null coalescing operator **??** to assign a default value to **Stranger** if it is null.

```
// Try to use null coalescing operator(??) to assign a default value to
Stranger if it is null
void main() {
  String? name;
  name = null;
  String name1 = name;
  print(name1);
}
```

Exercise 8: Type Promotion

Solve the error using type promotion:

```
// Try to solve the error using type promotion
Object name = "Mark";
print("The length of name is ${name.length}");
```

Exercise 9: Type Promotion

Solve the error using type promotion:

```
// Try to solve the error using type promotion
```

```dart
import 'dart:math';
class DataProvider{
    String? get stringorNull => Random().nextBool() ? "Hello" : null;

    void myMethod(){
        if(stringorNull is String){
            print("The length of value is ${stringorNull.length}");
        }else{
            print("The value is not string.");
        }

    }
}

void main() {
    DataProvider().myMethod();
}
```

Exercise 10: Late Keyword

Try using **late** keyword to solve the error:

```dart
// Try to solve the error using late keyword
class Person{
    String _name;

    void setName(String name){
        _name = name;
    }

    String get name => _name;
}

void main() {
    Person person = Person();
    person.setName("Mark");
    print(person.name);
}
```

QUESTION FOR PRACTICE 7

Question For Practice 7

1. What is the purpose of the **?** operator in Dart null safety?
2. Create a **late** variable named **address**, assign a **US** value to it and print it.
3. How do you declare a nullable type in Dart null safety?
4. Write a program in a dart to create an age variable and assign a **null** value to it using **?**.
5. Write a function that accepts a nullable int parameter and returns 0 if the value is null using null coalescing operator **??**.

6. Write a function named **generateRandom()** in dart that randomly returns **100** or **null**. Also, assign a return value of the function to a variable named **status** that can't be null. Give status a default value of 0, if **generateRandom()** function returns null.

Dart Asynchronous Programming

This section will help you to learn about the asynchronous programming in Dart. Here you will learn the following topics:

- Asynchronous Programming in Dart,
- Future in Dart,
- Async and Await in Dart, and
- Stream in Dart.

Practice Questions

Complete this section & practice this question to improve and test your dart programming knowledge.

ASYNCHRONOUS PROGRAMMING

Asynchronous Programming In Dart

Asynchronous Programming is a way of writing code that allows a program to do multiple tasks at the same time. Time consuming operations like fetching data from the internet, writing to a database, reading from a file, and downloading a file can be performed without blocking the main thread of execution.

Synchronous Programming

In Synchronous programming, the program is executed line by line, one at a time. Synchronous operation means a task that needs to be solved before proceeding to the next one.

Example Of Synchronous Programming

```dart
void main() {
  print("First Operation");
  print("Second Big Operation");
  print("Third Operation");
  print("Last Operation");
}
```

Show Output

```
First Operation
Second Big Operation
Third Operation
Last Operation
```

Here in this example, you can see that it will print line by line. Let’s suppose **Second Big Operation** takes 3 seconds to load then **Third Operation** and **Last Operation** need to wait for 3 seconds. To solve this issue asynchronous programming is here.

Asynchronous Programming

In Asynchronous programming, program execution continues to the next line without waiting to complete other work. It simply means, **Don’t wait**. It represents the task that doesn’t need to solve before proceeding to the next one.

Note: Asynchronous Programming improves the responsiveness of the program.

Example Of Asynchronous Programming

```
void main() {
  print("First Operation");
  Future.delayed(Duration(seconds:3),()=>print('Second Big Operation'));
  print("Third Operation");
  print("Last Operation");
}
```

Show Output

```
First Operation
Third Operation
Last Operation
Second Big Operation
```

Here in this example, you can see that it will print **Second Big Operation** at last. It is taking 3 seconds to load and **Third Operation** and **Last Operation** don’t need to wait for 3 seconds. This is the problem solved by Asynchronous Programming. A Future represents a value that is not yet available, you will learn about Future in the next section.

Why We Need Asynchronous

- To Fetch Data From Internet,
- To Write Something to Database,
- To execute a long-time consuming task,
- To Read Data From File, and
- To Download File etc.

Such **asynchronous operations** usually take a long time to complete, so it usually provide results in the form of a **Future**. If the result has multiple parts, then it provides as a **Stream**. You will learn about Future and Stream in the next section.

Note: To Perform asynchronous operations in dart you can use the **Future** class and the **async** and **await** keywords. We will learn Future, Async, and Await later in this guide.

Important Terms

- **Synchronous** operation blocks other operations from running until it completes.
- **Synchronous** function only perform a synchronous operation.
- **Asynchronous** operation allows other operations to run before it completes.
- **Asynchronous** function performs at least one asynchronous operation and can also perform synchronous operations.

FUTURE IN DART

Future In Dart

In dart, the Future represents a value or error that is not yet available. It is used to represent a potential value, or error, that will be available at some time in the future.

How To Create Future In Dart

You can create a future in dart by using **Future** class. Here the function will return `Future<String>` after 5 seconds.

```
// function that returns a future
Future<String> getUserName() async {
  return Future.delayed(Duration(seconds: 2), () => 'Mark');
}
```

You can also create a future by using `Future.value()` method. Here the function will return `Future<String>` immediately.

```
// function that returns a future
Future<String> getUserName() {
  return Future.value('Mark');
}
```

How To Use Future In Dart

You can use future in dart by using `then()` method. Here the function will return `Future<String>` after 5 seconds.

```
// function that returns a future
Future<String> getUserName() async {
  return Future.delayed(Duration(seconds: 2), () => 'Mark');
}

// main function
void main() {
  print("Start");
  getUserName().then((value) => print(value));
  print("End");
}
```

Show Output

```
Start
End
Mark
```

More About Future

Future represents the result of an asynchronous operation and can have 2 states.

State Of Future

- **Uncompleted**
- **Completed**

Uncompleted

When you call an asynchronous function, it returns to an uncompleted future. It means the future is waiting for the function asynchronous operation to finish or to throw an error.

Completed

It can be completed with value or completed with error. `Future<int>` produces an int value, and `Future<String>` produces a String value. If the future doesn't produce any value, then the type of future is `Future<void>`.

Note: If the asynchronous operation performed by the function fails due to any reason, the future completes with an error.

Example 2: Future In Dart

In this example below, we are creating a function **middleFunction()** that returns a future. The function will return `Future<String>` after 5 seconds.

```
void main() {
  print("Start");
  getData();
  print("End");
}

void getData() async{
  String data = await middleFunction();
  print(data);
}

Future<String> middleFunction(){
  return Future.delayed(Duration(seconds:5), ()=> "Hello");
}
```

Show Output

```
Start
End
Hello
```

Note: In the above example, First, it prints **Start**, secondly it prints **End**, and after 5 seconds **Hello** will be printed.

ASYNC AND AWAIT IN DART

Async And Await In Dart

Async/await is a feature in Dart that allows us to write asynchronous code that looks and behaves like synchronous code, making it easier to read.

When a function is marked **async**, it signifies that it will carry out some work that could take some time and will return a Future object that wraps the result of that work.

The **await** keyword, on the other hand, allows you to delay the execution of an async function until the awaited Future has finished. This enables us to create code that appears to be synchronous but is actually asynchronous.

The **async** and **await** keywords both provide a declarative way to define an asynchronous function and use their results. You can use the **async** keyword before a function body to make it asynchronous. You can use the **await** keyword to get the completed result of an asynchronous expression.

Important Concept

- To define an Asynchronous function, add async before the function body.
- The await keyword work only in the async function.

Example 1: Synchronous Function

```
void main() {
  print("Start");
  getData();
  print("End");
}

void getData() {
  String data = middleFunction();
  print(data);
}

Future<String> middleFunction(){
  return Future.delayed(Duration(seconds:5), ()=> "Hello");
}
```

Show Output

```
Start
End
Instance of '_Future<String>'
```

Example 2: Asynchronous function

```
void main() {
  print("Start");
  getData();
  print("End");
}

void getData() async{
  String data = await middleFunction();
  print(data);
}

Future<String> middleFunction(){
  return Future.delayed(Duration(seconds:5), ()=> "Hello");
}
```

Show Output

```
Start
End
Hello
```

In the above example, `async` handles the states of the program where any part of the program can be executed.`async` always comes with `await` because `await` holds the part of the program until the rest of the program executed.

Handling Errors

You can handle errors in the dart async function by using `try-catch`. You can write try-catch code the same way you write synchronous code.

Example 3: Handling Errors

```
main() {
  print("Start");
  getData();
  print("End");
}


void getData() async{
    try{
        String data = await middleFunction();
        print(data);
    }catch(err){
        print("Some error $err");
    }

}

Future<String> middleFunction(){
  return Future.delayed(Duration(seconds:5), ()=> "Hello");
}
```

Show Output

In the above example, `try-catch` handles the exception that could come after the program is executed.

Note: We cannot perform an asynchronous operation from a synchronous function.

Important Terms

- **async** The async keyword can be used before a function's body to indicate that a function is asynchronous.
- **async function** Functions marked with the async keyword are known as async functions.
- **await** The completed output of an asynchronous expression can be retrieved with the await keyword. Only async functions can use the await keyword.

STREAMS IN DART

A stream is a sequence of asynchronous events representing multiple values that will arrive in the future. Stream class deals with sequences of events instead of single events. Stream has one or more listeners, and all listeners will receive the same value.

For example, A stream is like a pipe that emits events, you put a value on the one end, and if there's a listener on the other end that listener will receive that value. These events can be values of any type, errors or a "done" event to signal the end of the stream.

	Single Value	Zero or more values
Sync	`int`	`Iterator`
Async	`Future<int>`	`Stream<int>`

How To Create Stream In Dart

You can create a stream in dart by using **Stream** class. Here the function will return `Stream<String>` after 5 seconds.

```
// function that returns a stream
Stream<String> getUserName() async* {
  await Future.delayed(Duration(seconds: 1));
  yield 'Mark';
  await Future.delayed(Duration(seconds: 1));
  yield 'John';
  await Future.delayed(Duration(seconds: 1));
  yield 'Smith';
}
```

Note: Here **yield** returns the value from the stream. To use **yield** you have to use `async*`.

You can also create a stream by using `Stream.fromIterable()` method. Here the function will return `Stream<String>` immediately.

```
// function that returns a stream
Stream<String> getUserName() {
  return Stream.fromIterable(['Mark', 'John', 'Smith']);
}
```

How To Use Stream In Dart

You can use stream in dart by using `await for` loop.

```
// function that returns a stream
Stream<String> getUserName() async* {
  await Future.delayed(Duration(seconds: 1));
  yield 'Mark';
  await Future.delayed(Duration(seconds: 1));
  yield 'John';
  await Future.delayed(Duration(seconds: 1));
  yield 'Smith';
```

Future	Stream
Future represents the value or error that is supposed to be available in the Future.	Stream is a way by which we receive a sequence of events.
A Future can provide only a single result over time.	Stream can provide zero or more values.
You can use FutureBuilder to view and interact with data.	You can use StreamBuilder to view and interact with data.
It can't listen to a variable change.	But Stream can listen to a variable change.
Syntax: Future <data_type> class_name	Syntax: Stream <data_type> class_name

```
}

// main function
void main() async {
  // you can use await for loop to get the value from stream
  await for (String name in getUserName()) {
    print(name);
  }
}
```

Show Output

```
Start
End
Mark
John
Smith
```

Future vs Stream

Types Of Stream

There are two types of streams:

1. Single Subscription streams
2. Broadcast streams

Single Subscription Stream

By default, Streams are set up for a single subscription. They hold onto the values until someone subscribes and can only be listened to once. You will get an exception if you try to listen more than once. Any event's value should not be missed and must be in the correct order. Inside the stream controller, there is only one stream, and only one subscriber can use that stream.

Broadcast Stream

This is the stream that is set up for multiple subscriptions. They hold onto the values until subscribers can only listen many times. You can use the broadcast stream if you want more objects to listen to the stream. It can be used for mouse events in a browser. Inside the stream controller, many streams can be used by many subscribers. E.g., You can start watching videos on such a stream at any time, and more than one subscriber can watch the video simultaneously. Similarly, you can watch again after canceling a previous subscription.

Syntax

```
StreamController<data_type> controller =
StreamController<data_type>.broadcast();
```

How Streams Are Created

You can create a stream in many ways. Let's create a `StreamController` first.

```
StreamController<data_type> controller = StreamController<data_type>();
```

Now we can access this controller through the `stream` property.

```
Stream stream = controller.stream;
```

How To Subscribe A Stream

After getting access from the stream you subscribe to the stream by calling a `listen()` method.

```
 stream.listen((value) {
  print("Value from controller: $value");
});
```

How To Add Value To The Stream

We can add the stream by calling the `add()` method. Let's add some value to the stream.

```
controller.add(3);
```

When we call the above function, we'll get the output as:

Value from controller: 3

How To Manage The Stream

To manage the stream, `listen()` method is used.

```
StreamSubscription<int> streamSubscription = stream.listen((value){
  print("Value from controller: $value");
});
```

How To Cancel A Stream

You can cancel a stream by using the `cancel()` method.

```
streamSubscription.cancel();
```

Types Of Classes In Stream

Four major classes in Dart's async libraries are used to manage streams.

Stream: It represents an asynchronous stream of data. For E.g:

```
 final controller = StreamController<String>();

final subscription = controller.stream.listen((String data) {
  print(data);
});
controller.sink.add("Data!");
```

EventSink: It is like a stream that flows in the opposite direction.

StreamController: It simplifies stream management, automatically creating a stream and sink and also providing methods for controlling a stream's behavior.

StreamSubscription: It saves the references of the subscription and allows them to pause, resume or cancel the flow of data they receive.

Method Used In Stream

There are four methods used in the stream: *listen(): It returns a StreamSubscription object representing the active stream-producing events. The stream subscription allows you to pause, resume the subscription after a pause, and cancel the subscription completely.

Syntax: listen

```
final subscription = myStream.listen()
```

- onError: Stream can provide errors just like a future can; by adding an `onError` method, you can catch and process an mistakes.

Syntax: onError

```
onError: (err){

}
```

- cancelOnError: This property or method is true by default but can be set to false to keep the subscription going even after an error.

Syntax: cancelOnError

```
cancelOnError : false
```

- onDone: This method can execute some code when the stream is finished sending data, such as when a file has been completely read.

Syntax: onDone

```
onDone: (){

}
```

Keywords Used In Stream

- async*: It is mainly used in the stream that works like the async in the future.
- yield: It is used to emit values from a generator, either async or sync. yield returns values from an Iterable or a Stream.
- yield*: yield* is used to call its Iterable or Stream function recursively.

Example Of async

```
Future<int> doSomeLongTask() async {
  await Future.delayed(const Duration(seconds: 2));
  return 21;
}main() async {
  int result = await doSomeLongTask();
  print(result); // prints '42' after waiting 2 second
}
```

Show Output

Example Of async In Dart*

```
Stream<int> countForOneMinute() async* {
  for (int i = 1; i <= 5; i++) {
    await Future.delayed(const Duration(seconds: 1));
    yield i;
  }
} main() async {
  await for (int i in countForOneMinute()) {
    print(i); // prints 1 to 5, one integer per second
  }
}
```

Show Output

```
1
2
3
4
5
```

Example Of yield In Dart*

```
Stream<int> str(int n) async* {
 if (n > 0) {
   await Future.delayed(Duration(seconds: 2));
   yield n;
   yield* str(n - 2);
 }
}

void main() {
 str(10).forEach(print);
}
```

Show Output

```
10
8
6
4
2
```

In the above example, you have printed only an even number from 10 to 2 using stream. It will print the number after 2 sec.

Some More Example OF Stream

Example 1

```
import 'dart:async';

void main() {
  var controller = StreamController();
  controller.stream.listen((event) {
    print(event);
  });
  controller.add('Hello');
  controller.add(42);
  controller.addError('Error!');
  controller.close();
}
```

Show Output

```
Hello
42
Uncaught Error: Error!
```

In this example, a String, integer and an error are added to the `StreamController` and then printed using the listen property.

Example 2

```
Stream<int> numberOfStream(int number) async* {
  for (int i = 0; i <= number; i++) {
    yield i;
  }
}

void main(List<String> arguments) {
  // Calling the Stream
  var stream = numberOfStream(6);
  // Listening to Stream yielding each number
  stream.listen((s) => print(s));
}
```

Show Output

```
0
1
2
3
4
5
6
```

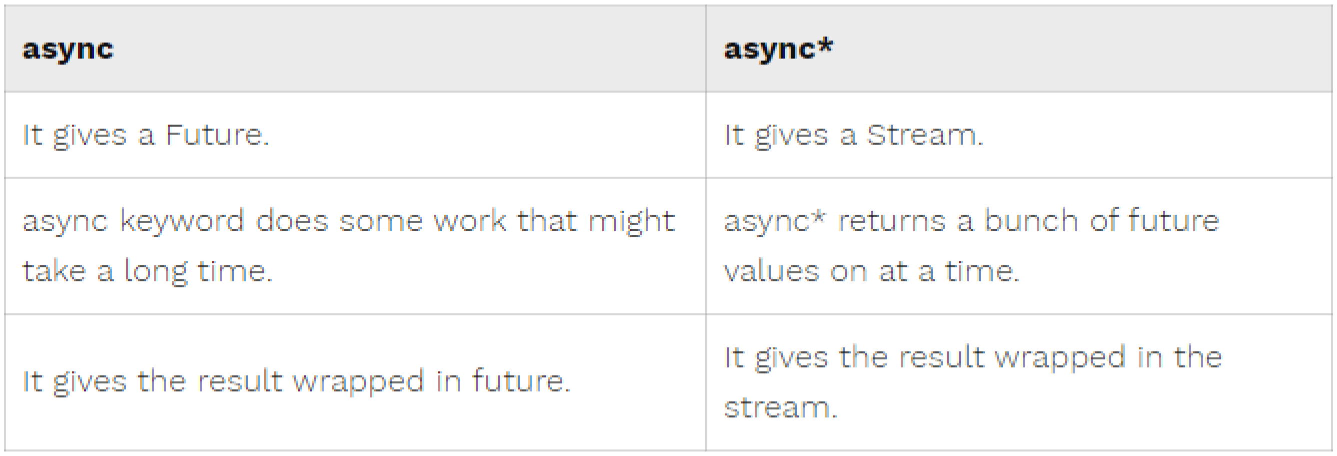
In the above example, you must print the number from 0 to 6 using stream.

Example 3

```
Stream<int> str(int n) async* {
 for (var i = 1; i <= n; i++) {
   await Future.delayed(Duration(seconds: 1));
   yield i;
 }
}

void main() {
 str(10).forEach(print);
}
```
Show Output

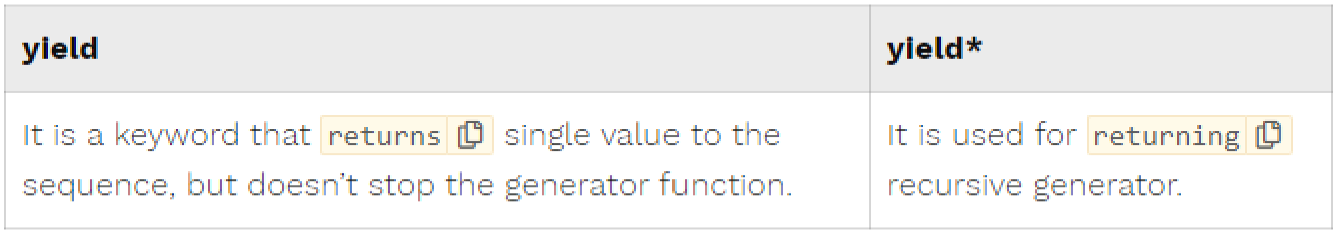
```
1
2
3
4
5
```

In the above example, you must print the number from 1 to 5 using stream. It will print the number after 1 sec.



async	async*
It gives a Future.	It gives a Stream.
async keyword does some work that might take a long time.	async* returns a bunch of future values on at a time.
It gives the result wrapped in future.	It gives the result wrapped in the stream.

async vs async In Dart*



yield	yield*
It is a keyword that `returns` single value to the sequence, but doesn't stop the generator function.	It is used for `returning` recursive generator.

*yield vs yield In Dart**

To sum up, Streams are used in Dart to handle asynchronous data flows. They allow us to process data as it becomes available, rather than waiting for it to be fully loaded before processing.

Streams are commonly used in scenarios where data is being continuously updated or where we want to handle events as they occur. For example, we can use streams to monitor user interactions in real-time, or to receive data from a server as it becomes available.

In Dart, we can use the Stream and StreamController classes to create and manage streams. The StreamController class is used to create a stream and add data to it, while the Stream class is used to listen to the stream and process incoming data.

Ultimately, streams are a strong feature in Dart that let us handle asynchronous data flows in a flexible and effective way.

QUESTION FOR PRACTICE 8

Dart Asynchronous Programming Practice Questions

1. Explain what is `asynchronous programming` in dart?
2. What is `Future` in dart?
3. Write a program to print current time after 2 seconds using `Future.delayed()`.
4. Write a program in dart that reads csv file and print it's content.
5. Write a program in dart that uses Future class to perform multiple asynchronous operations, wait for all of them to complete, and then print the results.
6. Write a Dart program to calculate the sum of two numbers using async/await.
7. Write a Dart program that takes in two integers as input, waits for 3 seconds, and then prints the sum of the two numbers.
8. Write a Dart program that takes a list of strings as input, sorts the list asynchronously, and then prints the sorted list.
9. Write a Dart program that takes a list of integers as input, multiplies each integer by 2 asynchronously, and then prints the modified list.
10. Write a Dart program that takes a string as input, reverses the string asynchronously, and then prints the reversed string.

Dart Useful Information

This chapter will teach you the useful information about Dart programming language. Here you will learn the following topics:

- Final and Const in Dart,

- Date and Time in Dart, and
- Extension in Dart.

FINAL VS CONST

Final Vs Const In Dart

If you do not want to change the value of a variable, then you can use either final or const in dart.

Example

```
void main() {
  final finalName = "Final John Doe";
  const constName = "Const John Doe";

  finalName = "Raj"; // Not Possible
  constName = "Anu"; // Not Possible

  print("Final name is " + finalName);
  print("Const name is " + constName);
}
```

Show Output

```
Error: Can't assign to the final variable 'finalName'.
Error: Can't assign to the const variable 'constName'.
```

Const In Dart

If you need to calculate value at compile-time, it is a good idea to choose `const` over `final.` A const variable is a compile-time constant. They must be created from data that can be calculated at compile time. `100+1` is valid const expression but `const date = DateTime.now();` is not.

What Is Compile Time

When you run code in the dart, it will be `compiled` into the format that the machine can understand. This time is called compile time. Const value should be known at compile time.

What Is Run Time

Runtime is the time when your compiled code is started running. It generally occurs after the compile time.

Note: If you use `const` inside the class, declare it as `static const.`

```
const total = 50+50; // Possible
const date = DateTime.now(); // Not Possible
```

Advantage Of Constant

- Improve Performance

Final In Dart

If the value is calculated at runtime, you can choose final for it. For. e.g if you want to calculate date on run time, you can use `final date = DateTime.now();` but not `const date = DateTime.now();`.

Note: Anything that is unknown at compile time should be `final` over `const.`

```
final date = DateTime.now(); // Possible
const date = DateTime.now(); // Not Possible
```

When To Use Const

- If you know the value at compile-time, choose `const` for e.g. `const a = 100;`.

When To Use Final

- If you don't know the value at compile-time, choose `final`.
- If you want a network request that can't be changed, choose `final`.
- If you want to get some values from the database, choose `final`.
- If you want to read a local file, choose `final`.

Note: Final variables will have a value known at runtime. Const variables have a value known at compile time. Instance variable can be final but not const.

DATETIME IN DART

DateTime In Dart

Date and time are often used in our day-to-day activities. As a programmer you need to know how to find a date and time? How to format date? and how to perform different calculation in date?

How To Get Date And Time

Use the following code to get the current date and time in the dart.

```
void main() {
  print(DateTime.now());
}
```

Get Year, Month, Day Of Datetime In Dart

Here is the way to get a year, month, day, hour, minutes, and seconds in Dart. You can convert DateTime to String by using the `toString()` method.

Example

```
void main() {
  DateTime datetime = DateTime.now();
  print("Year is " + datetime.year.toString());
  print("Month is " + datetime.month.toString());
  print("Day is ${datetime.day}"); // If you don't want to use .toString
  print("Hour is " + datetime.hour.toString());
  print("Minutes is " + datetime.minute.toString());
  print("Second is " + datetime.second.toString());
}
```

How To Convert Datetime To String In Dart

Use the following code to convert DateTime to String in the dart.

```
void main() {
  String datetime = DateTime.now().toString();
  print(datetime);
}
```

How To Convert String To DateTime

You cannot get year, months, or day directly and cannot perform date calculation using a String if that String contains the correct DateTime value. In such a situation, you first need to convert String to DateTime.

```
void main() {
  String myDateInString = "2022-05-01";
  DateTime myConvertedDate = DateTime.parse(myDateInString);
  print("Year is " + myConvertedDate.year.toString());
  print("Month is " + myConvertedDate.month.toString());
  print("Day is " + myConvertedDate.day.toString());
}
```

Methods Supported By Datetime In Dart

You can use DateTime methods if you want to add days, hours, or minutes to DateTime. Let us suppose you have created a DateTime object named mybirthday. `DateTime mybirthday = DateTime.parse("1997-05-14");`

Method	Example
add(Duration)	myBirthday.add(Duration(days: 1));
subtract(Duration)	myBirthday.subtract(Duration(days: 1));

Note: You can set a duration to `days`, `hours`, `minutes`, `seconds`, `milliseconds`, and `microseconds`. To understand it more, look at the example below.

Example: Add Date In Dart

```
void main() {
  DateTime myBirthday = DateTime.parse("1997-05-14");
  myBirthday = myBirthday.add(Duration(days: 1));
  print("Year is " + myBirthday.year.toString());
  print("Month is " + myBirthday.month.toString());
  print("Day is " + myBirthday.day.toString());
}
```

Example: Subtract Date In Dart

```
void main() {
  DateTime myBirthday = DateTime.parse("1997-05-14");
  myBirthday = myBirthday.subtract(Duration(days: 1));
  print("Year is " + myBirthday.year.toString());
  print("Month is " + myBirthday.month.toString());
  print("Day is " + myBirthday.day.toString());
}
```

Find Difference Between Two Dates In Dart

Suppose you want to find the difference between two dates in dart. There is a straightforward way.

```
void main() {
  DateTime myBirthday = DateTime.parse("1997-05-14");
  DateTime today = DateTime.now();
  Duration diff = today.difference(myBirthday);
  print("Difference in days: " + diff.inDays.toString());
  print("Difference in hours: " + diff.inHours.toString());
  print("Difference in minutes: " + diff.inMinutes.toString());
  print("Difference in seconds: " + diff.inSeconds.toString());
  print("Difference in milliseconds: " + diff.inMilliseconds.toString());
  print("Difference in microseconds: " + diff.inMicroseconds.toString());
}
```

Name	Description
inDays	Convert duration in days.
inHours	Convert duration in hours.
inMinutes	Convert duration in minutes.
inSeconds	Convert duration in seconds.
inMilliseconds	Convert duration in milli seconds.
inMicroseconds	Convert duration in micro seconds.

DateTime Comparision Methods

If you want to compare two dates, then you can use comparison methods.

Method Name	Description
IsAfter(DateTime)	Returns true or false. `bool`
IsBefore(DateTime)	Returns true or false. `bool`
IsAtTheSameMoment(DateTime)	Returns true or false. `bool`

```
void main() {
  DateTime myBirthday = DateTime.parse("1997-05-14");
  DateTime today = DateTime.now();

  if (myBirthday.isBefore(today)) {
    print("My Birthday is before today.");
  } else if (myBirthday.isAfter(today)) {
    print("My Birthday is after today.");
  } else if (myBirthday.isAtSameMomentAs(today)) {
    print("My Birthday date and today's date is same.");
  }

}
```
Show Output

```
My Birthday is before today.
```

EXTENSION IN DART

Dart Extension Method

In Dart, you can extend the functionality of a class by using extension. It is a new feature in Dart 2.7.0. It is similar to extension methods in C# and Kotlin. It is also similar to the concept of mixins in Dart.

How To Use Extension In Dart

Here we are extending the functionality of **String** class. We are adding a new method **capitalize** to the **String** class. We are using **extension** keyword to extend the functionality of **String** class.

```
void main(){
  String name = "john";
  print(name.capitalize());
}

extension StringExtension on String{
  String capitalize(){
    return "${this[0].toUpperCase()}${this.substring(1)}";
  }
}
```
Show Output

```
John
```

BACKEND IN DART

Dart Backend

Backend means an application runs on a server and can handle client requests. It can accept requests from a client, process them and respond to the client. Currently, many technologies provide a backend to clients, such as golang, node js, PHP, python, java, .net, etc.

Why Backend Framework Needed On Dart

There are already various backend services, but why do you need a backend in dart? The main reason is **Flutter**. With Flutter, you can create apps for android, ios, web, and desktop with a single codebase. Flutter is loved by many developers and gained so much popularity. Now, most of the developers already know dart. Why should they need to learn another programming language for the backend? What if they can do it by dart programming?

Here are some other reasons why we need a framework in dart: 2. Rich package library. 3. Hard to do everything from the beginning, like architecture design and guidelines.

Backend For Dart

Here is a possible backend for the dart.

- Shelf,
- Dart Frog,
- ServerPod,
- Alfred,
- Get Server,
- ServeMe,
- Angel3 Framework,
- Lucifer.

DART INTERVIEW QUESTIONS

Dart Interview Questions

Here are the most frequently asked Dart interview questions. These Dart interview questions are suitable for both freshers and experienced programmers. These Dart interview questions are designed to test your knowledge of Dart programming language.

Basic Dart Interview Questions

- What is Dart?
- What inspired you to pursue a career in dart programming?

- What are your favorite aspects of dart programming?
- What do you think sets dart programming apart from other languages?
- What is your favorite dart programming feature?
- What do you think is the most challenging part of dart programming?
- What do you think would be the biggest challenge facing dart programmers in the future?
- What do you think is the most important thing for dart programmers to remember?
- What do you think is the best thing about dart programming?

Dart How To?

In this section, you will learn how to ideas for dart and make your programming life more effortless and exciting with this section.

This will include following topics:

- Generate Random Number
- Convert String To Number
- Capatilize First Letter Of String
- Make http Request
- Reverse List

CONVERT STRING TO INT

Convert String To Int In Dart

The String is the textual representation of data, whereas int is a numeric representation without a decimal point. While coding in must of the case, you need to convert data from one type to another type. In dart, you can convert String to int by using the **int.parse() method**.

```
int.parse("String");
```

You can replace **String** with any numeric value and convert String to int. Make sure you have numeric value there. To know more see the example below.

Example To Convert String to Int

This program converts String to int. You can view type of variable with **.runtimeType**.

```
void main(){
String value = "10";
int numericValue = int.parse(value);
print("Type of value is ${value.runtimeType}");
print("Type of numeric value is ${numericValue.runtimeType}");
}
```

Show Output

```
Type of value is String
Type of numeric value is int
```

Example 2 To Convert String to Int

This program first converts String to int and finds the sum of two numbers.

```
void main(){
String value = "10";

int num2 = 20;
int num1 = int.parse(value);

int sum = num1 + num2;

print("Type sum is $sum");
}
```

Show Output

```
Sum is 110
```

Things To Remember While Casting String To Int

You must be sure that String could convert the to int. If it doesn't convert to int, it will throw an exception. You can use the try-catch statement to show your custom message. If you don't know exception handling in dart learn from here.

Example With Try Catch

This program throws an exception because you can't convert "hello" to int. In this situation, you can use the try-catch exception to display your custom message.

```
void main(){
try {
String value = "hello";
int numericValue = int.parse(value);
print("Type of numeric value is ${numericValue.runtimeType}");

  }
catch(ex){
print("Something went wrong.");
}

}
```

Show Output

```
Something went wrong.
```

GENERATE RANDOM NUMBER

Generate Random Number In Dart

This tutorial will teach you how to generate random numbers in dart programming. You will learn to generate random numbers between a range, and also generate a list of random numbers.

Why You Need To Generate Random Number

- **Random Number Game**: You can use random numbers to create a random number game.
- **Card Game**: You can use random numbers to shuffle the cards.

Example 1: Generate Random Number In Dart

This example shows how to generate random numbers from **0 - 9** and also **1 to 10**. After watching this example, you can generate a random number between your choices.

```
import 'dart:math';
void main()
{
Random random = new Random();
int randomNumber = random.nextInt(10); // from 0 to 9 included
print("Generated Random Number Between 0 to 9: $randomNumber");

int randomNumber2 = random.nextInt(10)+1; // from 1 to 10 included
print("Generated Random Number Between 1 to 10: $randomNumber2");
}
```

- In this program, **random.nextInt(10)** function is used to generate a random number between **0 and 9** in which the value is stored in a variable **randomNumber**.
- The **random.nextInt(10)+1** function is used to generate random number between **1 to 10** in which the value is stored in a variable **randomNumber2**.

Generate Random Number Between Any Number

Use this formula to generate a random number between any numbers in the dart.

```
 min + Random().nextInt((max + 1) - min);
```

Example 2: Random Number In Dart Between 10 - 20

This program generates random numbers between 10 to 20.

```
import 'dart:math';
void main()
```

```
{

int min = 10;
int max = 20;

int randomnum = min + Random().nextInt((max + 1) - min);

print("Generated Random number between $min and $max is: $randomnum");
}
```

CAPITALIZE FIRST CHARACTER

How To Capitalize First Letter Of String In Dart

If you want to capitalize the first letter of a String in Dart, you can use the following code:

```
//Example of capitalize first letter of String
void main() {
  String text = "hello world";
  print("Capitalized first letter of String:
${text[0].toUpperCase()}${text.substring(1)}");
}
```

Show Output

```
Capitalized first letter of String: Hello world
```

Example 2: To Capitalize First Letter Of String Using Extension Method

In this example, we will use the extension method to capitalize the first letter of a String. You can learn more about extension method here.

```
//Example of capitalize first letter of String using extension method
extension StringExtension on String {
  String capitalize() {
    return "${this[0].toUpperCase()}${this.substring(1)}";
  }
}

void main() {
  String text = "hello world";
  print("Capitalized first letter of String: ${text.capitalize()}");
}
```

Show Output

```
Capitalized first letter of String: Hello world
```

MAKE HTTP REQUEST IN DART

Introduction

HTTP request means sending a request to a server and getting a response from the server. In this tutorial, we will learn how to make http request in dart with examples.

Create Dart Project

Before starting this tutorial, you need to create a dart project. Make sure you have installed the dart sdk in your system. First, open the command prompt/terminal and type the following command to create a dart project.

```
dart create <project_name>
```

This will create a simple dart project with some ready-made code.

Steps To Create Dart Project

- Open folder location on command prompt/terminal.
- Type `dart create my_app`
- Type `cd my_app`
- Type `code .` to open project with visual studio code
- To check main dart file go to **bin/my_app.dart** and edit your code.

Run Dart Project

First, open the project location on the command/terminal and run the project with this command.

```
dart run
```

Download Http Package

To make HTTP request in dart, we need to download the http package. To download the http package, open the pubspec.yaml file and add the following line in the dependencies section. You can get http package from here.

```
dependencies:
  http: ^0.13.5
```

Make HTTP Get Request

When you want to get data from the server, you need to make a get request. To make HTTP get request in dart, you can use get() method on the http client instance.

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


void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  // make http get request
  var response = await http.get(url);
  // check the status code for the result
  if (response.statusCode == 200) {
    print(response.body);
  } else {
    print('Request failed with status: ${response.statusCode}.');
  }

}
```

Note: If the status code is 200, it means the request is successful and you can get the response body otherwise the request is failed and you can get error message.

Make HTTP Post Request

When you want to send data to the server, you need to make a post request. To make HTTP post request in dart, you can use post() method on the http client instance.

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

void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts');
  // make http post request
  var response = await http.post(url, body: {'title': 'foo', 'body':
'bar', 'userId': '1'});
  // check the status code for the result
  if (response.statusCode == 201) {
    print(response.body);
  } else {
    print('Request failed with status: ${response.statusCode}.');
  }

}
```

Make HTTP Put Request

When you want to update data on the server, you need to make a put request. To make HTTP put request in dart, you can use put() method on the http client instance.

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

void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  // make http put request
  var response = await http.put(url, body: {'title': 'foo', 'body': 'bar',
'userId': '1'});
  // check the status code for the result
  if (response.statusCode == 200) {
    print(response.body);
  } else {
    print('Request failed with status: ${response.statusCode}.');
  }

}
```

Make HTTP Patch Request

When you want to update a part of data on the server, you need to make a patch request. To make HTTP patch request in dart, you can use patch() method on the http client instance.

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

void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  // make http patch request
  var response = await http.patch(url, body: {'title': 'foo'});
  // check the status code for the result
  if (response.statusCode == 200) {
    print(response.body);
  } else {
    print('Request failed with status: ${response.statusCode}.');
  }

}
```

Make HTTP Delete Request

When you want to delete data from the server, you need to make a delete request. To make HTTP delete request in dart, you can use delete() method on the http client instance.

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

```

```
void main() async {
  var url = Uri.parse('https://jsonplaceholder.typicode.com/posts/1');
  // make http delete request
  var response = await http.delete(url);
  // check the status code for the result
  if (response.statusCode == 200) {
    print(response.body);
  } else {
    print('Request failed with status: ${response.statusCode}.');
  }

}
```

Note: If the status code is 200, it means the request is successful and you can get the response body otherwise the request is failed and you can get error message.

REVERSE A LIST IN DART

Reverse a List In Dart

There are several ways to reverse a list in dart. Here are some of the most common approaches:

Using Method

The **reversed** method returns an iterable that provides the reversed view of the list. You can convert this iterable to a list using the toList method. Here's an example:

```
void main(){
List<int> numbers = [1, 2, 3, 4, 5];
List<int> reversedNumbers = numbers.reversed.toList();
print(reversedNumbers); // [5, 4, 3, 2, 1]
}
```

Show Output

```
[5, 4, 3, 2, 1]
```

Using List.from Constructor

You can use the List.from constructor to create a new list from the original list and then call the reversed method on it. Here's an example:

```
void main(){
List<int> numbers = [1, 2, 3, 4, 5];
List<int> reversedNumbers = List.from(numbers.reversed);
print(reversedNumbers); // [5, 4, 3, 2, 1]
}
```

Show Output

```
[5, 4, 3, 2, 1]
```

Using a Loop

You can use a loop to iterate through the original list and add its elements to a new list in reverse order. Here's an example:

```
void main(){
List<int> numbers = [1, 2, 3, 4, 5];
List<int> reversedNumbers = [];
for (int i = numbers.length - 1; i >= 0; i--) {
  reversedNumbers.add(numbers[i]);
}

print(reversedNumbers); // [5, 4, 3, 2, 1]
}
```

Show Output

```
[5, 4, 3, 2, 1]
```

Note: All of these approaches produce the same result, which is a new list that contains the elements of the original list in reverse order. You can choose the one that suits your needs and preferences best.

Quiz Game

Dart Quiz By Technology Channel

Take our quiz and test your dart skill. Click on link to get started.

- Start Basic Dart Quiz

Start Basic Dart Quiz

1.What is Dart?*

Dart is a object-oriented programming language
Dart is used to create a frontend user interfaces
Both of the above
None of the above

2.What will be the output of this program :

void main() {   int num;   print(num); }*
Error
Null
Num

None of these

3.Which of these is not a keyword in dart?*

factory
yield
export
scan

4.Which framework uses dart?*

Python
Java
Flutter
React

5.Dart is an?*

Open-source
Asynchronous
Programming language
All of the above

6.Dart is originally developed by?*

Microsoft
Google
IBM
Facebook

7.The ________ function is a predefined method in Dart.*

declare()
list()
main()
return()

8.Dart is an Object-Oriented language.*

Yes
No

9. _________ is a real-time representation of any entity.*

Class
Method
Object
None of the above

10.What is the extension of Dart file?*

.dart
.py
.java
.drt

11. Which of the following statements does not use string interpolation correctly?*

print('Your name in upper case is $"me".toUpperCase');
print("Your name in upper case is ${'me'.toUpperCase()}");

```
print('Your name in upper case is ${"me".toUpperCase()}');
```

12.Which keyword in Dart is used to create a subclass*

extends
subclass
super
none of these

13.Instance variables in Dart cannot be*

final
const
Both of these
None of these

14.The print() method in Dart takes*

a single argument
two argument
multiple argument
user defined multiple arguments

15.Which command is used to run dart file "technologychannel.dart"?*

run technologychannel.dart
d technologychannel.dart
dart technologychannel.dart
start technologychannel.dart

16.What is commonly known as a dictionary or hash*

set
list
map
none of these

17.Is Dart Case Sensitive Programming Language?*

No
Yes

18.--version command is used to ?*

Enables assertions
Displays version information
Specifies the path
Specifies where to find imported libraries

19.The await keyboard works in*

async function only
sync function only
both async and sync function
none

20.The package import pub commands is*

pub get
pub delete

pub post
pub set

DART CONCLUSION

Congratulations

Congratulations on completing the dart course. We hope you enjoyed our content and improved your dart skill to the next level. Now you can use this concept to build apps with flutter confidently.

You can come back to this dart tutorial whenever you face difficulty while working.

Also if you like this work, please do share it with your friends. Your support can help us to provide other tutorials in the future. Also, go to the next section and check your dart knowledge by our **Quiz**.

What After Dart

After learning dart, it’s time to Learn Flutter, which is the most Popular App Development Framework. It will help you build high-quality android, iOS, and web apps from a single code base.

Advantage Of Flutter

- 1 Codebase For Android, iOS, Web, and Desktop,
- Huge Community Support,
- Easy to Learn,
- Use dart as a programming language.