# My Dart Documentation Summary

**1.**

**INSTALLING DART SDK AND INTRODUCTION**

If You have flutter SDK that is or above version 1.21, You already have dart SDK but to install Dart SDK separately:

**On Window:**

1. Click on the the link([**https://dart.dev/get-dart**](https://dart.dev/get-dart) )
2. scroll down and click on **downloading the SDK as a zip file**.
3. In this new page, click on **Dart SDk (SHA 256)** in the table under the heading **“stable channel”,** if your PC is 64bits.
4. After downloading, go to your C drive and create a new folder called anything you like but I will call mine **src**, so I now have:

**c:/src**

1. Now go to your downloads and extract the **dart zip** file to the **c:/src** you just created now.
2. After the extraction, go to **bin** folder inside the dart sdk folder you just extracted now, copy the path.
3. Go to your **pc search** and type **envir**, then click to **edit your environmental variable**
4. Then click **Environmental variable**…
5. under **users variable**, edit your **path** variable if you already have a **path** variable created before.

or else

click on **new**

then type **path** as **variable name**

and

paste the **path** you copied at **no. 6** above as **variable value**.

1. Then press **enter** key on the keyboard to **save** the changes.
2. After you must have saved and closed the new changes in your environmental variable, Open your command line and type:

* **dart** then hit **enter** button to see some code to shows that you have successfully installed dart.
* **dart --version** then hit **enter** button to see some code to shows the version of dart you installed.

## Important concepts:

***NB: Anything you don’t understand here, read-on because a better explanation is in later chapters.***

As you learn about the Dart language, keep these facts and concepts in mind:

* Everything you can place in a variable is an object, and every object is an instance of a class. Even numbers, functions, and null are objects. With the exception of null (if you enable [sound null safety](https://dart.dev/null-safety)), all objects inherit from the [Object](https://api.dart.dev/stable/dart-core/Object-class.html) class.
* **Version note:** [**Null safety**](https://dart.dev/null-safety) was introduced in **Dart 2.12** thus using null safety requires a [**language version**](https://dart.dev/guides/language/evolution#language-versioning) of at least **2.12**.
* Although Dart is strongly typed, type annotations are optional because Dart can infer types.
* If you enable [null safety](https://dart.dev/null-safety), variables can’t contain null unless you say they can. You can make a variable nullable by putting a question mark **(?)** at the end of its type. For example, a variable of type **int?** might be an integer, or it might be null. If you know that an expression never evaluates to null but Dart disagrees, you can add **(!)** to **assert** that it isn’t null (and to throw an exception if it is). An example: int x = nullableButNotNullInt!;
* When you want to explicitly say that any type is allowed, use the type Object? (if you’ve [enabled null safety](https://dart.dev/null-safety#enable-null-safety)), Object, or — if you must defer type checking until runtime — use the [special type dynamic](https://dart.dev/guides/language/effective-dart/design#avoid-using-dynamic-unless-you-want-to-disable-static-checking).
* Dart supports generic types, like List<int> (a list of integers) or List<Object> (a list of objects of any type).
* Dart supports top-level functions (such as main()), as well as functions tied to a class or object (static and instance methods, respectively). You can also create functions within functions (nested or local functions).
* Similarly, Dart supports top-level variables, as well as variables tied to a class or object (static and instance variables). Instance variables are sometimes known as **fields** or **properties**.
* Unlike Java, Dart doesn’t have the keywords public, protected, and private. If an identifier starts with an underscore **(\_)**, it’s private to its library.
* Identifiers can start with a letter or underscore (\_), followed by any combination of characters ,which digits can be inclusive.
* Dart has both expressions (which have runtime values) and statements (which don’t). For example, the [conditional expression](https://dart.dev/guides/language/language-tour#conditional-expressions) condition ? expr1 : expr2 has a value of expr1 or expr2. Compare that to an [if-else statement](https://dart.dev/guides/language/language-tour#if-and-else), which has no value. A statement often contains one or more expressions, but an expression can’t directly contain a statement.
* Dart tools can report two kinds of problems: **warnings** and **errors**. Warnings are just indications that your code might not work, but they don’t prevent your program from executing. Errors can be either **compile-time** or **run-time**. A compile-time error prevents the code from executing at all; a run-time error results in an [exception](https://dart.dev/guides/language/language-tour#exceptions) being raised while the code executes.

**2.**

**CREATING, RUNNING, TESTING, COMPILING AND DEBUGGING A NEW DART PROJECT**

To create a dart project, recently we have 5 templates that can aid the creation of different dart projects. You can type **dart create -h** to see these templates. These templates are:

1. **console-simple:**

This is the default template for command line application project.

You can create it with

**>dart create --t console-simple project\_name.**

1. **console-full:**

This is the full template for command line application sample

You can create it as below

**>dart create --t console-full project\_name.**

1. **package-simple:**

This is a template that serves as a starting point for dart libraries and applications.

It is created like below

**>dart create --t package-simple project\_name.**

1. **server-shelf:**

Used for server app using “package:shelf”.

**>dart create --t server-shelf project\_name.**

1. **web-simple:**

This is a template that helps to create web apps that uses dart core libraries.

it is created as below

**>dart create --t web-simple project\_name.**

*NB: You can uses* ***ctrl-shift-p*** *to create a dart project in* ***visual studio code*** *if you install it and install the* ***latest dart extension****. After pressing* ***ctrl-shift-p***, ***type dart****, then* ***select Dart:new project****, then you will see a list for you to select from any of the above 5 template.*

*Also,* **project name** *for both* **dart** *and* **flutter** *must be in* ***lowercase*** *(separated by underscore).*

~~You can also create dart package/project manually by creating a project\_name.dart file, pubspec.yaml and analysis\_options.yaml file. Then in pubspec.yaml file, add the name, environment,… etc and run dart pub get.~~

**RUNNING YOUR DART PROGRAM:**

1. **In VSCode:**

* **crtl F5** makes you to **run** your dart code **without debugging**.
* **F5** makes you to **run** and **degug** your dart code.
* You can also click the **play** **button**
* You can also click the **run** command ontop of **main** function.
* You can use > **dart run** if you are already in the package file.
* you can use **dart run filename** (or library\_name) that contains the main function.

like:

> **dart run full/bin/full.dart**

***teach how to install VScode***.

1. **In Android Studio:**

**I am yet to write it down so prefer vscode for now.**

1. **.**

**TESTING YOUR DART PACKAGE:**

* You can type >**dart test** and then click Enter in your terminal.
* You can also type > **dart test test/package\_name.dart** and click ENTER to test the app.
* You use the **run** command ontop of the **main function** in **test library** or the **run** command ontop of each **test function** inside the **main function**.

**COMPILING DART PACKAGE:**

* **Native platform**: For apps targeting mobile and desktop devices, Dart includes both a Dart VM with just-in-time (JIT) compiler and an ahead-of-time (AOT) compiler (for producing machine code).
* **Web platform**: For apps targeting the web, Dart includes both a **development time** compiler (**dartdevc**) and a **production time** compiler (**dart2js**). Both compilers translate Dart into JavaScript.

Dart’s compiler technology lets you run code in different ways:

* + 1. **KERNEL BINARY (.dill):**

This is the language Dart VM understands, thus all source code are first compiled to it before the Dart VM can compile the code. Source codes are compiled to kernel binary using **common front end** (CFE) package.

Use ***>dart compile kernel* path*\_to\_your\_ main\_dart\_file*** to compile your package to kernel binary. A file with **.dill** extension will be produced.

Kernel binary contains **kernel abstract syntax tree** (Kernel **AST**), which is a dart intermediary language.

* + 1. **AOT:**
    2. **.exe :**

Type in >***dart compile exe path\_to\_your\_ main\_dart\_file***.

The file generated can be run directly by mare typing the file path in your terminal. Example:

**PS C:\src\practices\bin> full/bin/full.exe**

* + 1. **JIT:**

It is only used in the development phase.

**DEBUGGING YOUR DART PACKAGE:**

1. **In VSCode:**

* F5 makes you to debug your Dart code.
* You can click the debug command ontop of main function.

1. **Using Devtools:**

* Open a terminal and type in ***dart pub global activate devtools*** and hit the **Enter( Return↵)** button. This will help you to download devtools.
* Now read what is output at the terminal, this will direct you to where the **devtools** is in your folder.
* copy the path to your devtools and paste it in under **path**, which is in **system variable**, which is in **environmental variable**, which is in **advance settings**, which is in **control panel**.
* Then reopen a terminal and type in **devtools**, this will open a tab in your browser that you will **paste a link** below.
* Open another terminal and type in ***dart run --observe --pause-isolates-on-start***. When you press **enter**, this will give you a link to paste in the **above** **browser tab**.

*Caveat: Use* ***cd packageName( or projectName)*** *to be inside the package or this won’t work.*

* Lo and behold the long awaited devtools is b4 you.

1. **In Android Studio:**

**3.**

**KEYWORDS, OPERATORS AND ASSERT.**

**Keywords:**

|  |  |  |  |
| --- | --- | --- | --- |
| [abstract](https://dart.dev/guides/language/language-tour#abstract-classes) 2 | [else](https://dart.dev/guides/language/language-tour#if-and-else) | [import](https://dart.dev/guides/language/language-tour#using-libraries) 2 | [super](https://dart.dev/guides/language/language-tour#extending-a-class) |
| [as](https://dart.dev/guides/language/language-tour#type-test-operators) 2 | [enum](https://dart.dev/guides/language/language-tour#enumerated-types) | [in](https://dart.dev/guides/language/language-tour#for-loops) | [switch](https://dart.dev/guides/language/language-tour#switch-and-case) |
| [assert](https://dart.dev/guides/language/language-tour#assert) | [export](https://dart.dev/guides/libraries/create-library-packages) 2 | [interface](https://stackoverflow.com/questions/28595501/was-the-interface-keyword-removed-from-dart) 2 | [sync](https://dart.dev/guides/language/language-tour#generators) 1 |
| [async](https://dart.dev/guides/language/language-tour#asynchrony-support) 1 | [extends](https://dart.dev/guides/language/language-tour#extending-a-class) | [is](https://dart.dev/guides/language/language-tour#type-test-operators) | [this](https://dart.dev/guides/language/language-tour#constructors) |
| [await](https://dart.dev/guides/language/language-tour#asynchrony-support) 3 | [extension](https://dart.dev/guides/language/language-tour#extension-methods) 2 | [library](https://dart.dev/guides/language/language-tour#libraries-and-visibility) 2 | [throw](https://dart.dev/guides/language/language-tour#throw) |
| [break](https://dart.dev/guides/language/language-tour#break-and-continue) | [external](https://stackoverflow.com/questions/24929659/what-does-external-mean-in-dart) 2 | [mixin](https://dart.dev/guides/language/language-tour#adding-features-to-a-class-mixins) 2 | [true](https://dart.dev/guides/language/language-tour#booleans) |
| [case](https://dart.dev/guides/language/language-tour#switch-and-case) | [factory](https://dart.dev/guides/language/language-tour#factory-constructors) 2 | [new](https://dart.dev/guides/language/language-tour#using-constructors) | [try](https://dart.dev/guides/language/language-tour#catch) |
| [catch](https://dart.dev/guides/language/language-tour#catch) | [false](https://dart.dev/guides/language/language-tour#booleans) | [null](https://dart.dev/guides/language/language-tour#default-value) | [typedef](https://dart.dev/guides/language/language-tour#typedefs) 2 |
| [class](https://dart.dev/guides/language/language-tour#instance-variables) | [final](https://dart.dev/guides/language/language-tour#final-and-const) | [on](https://dart.dev/guides/language/language-tour#catch) 1 | [var](https://dart.dev/guides/language/language-tour#variables) |
| [const](https://dart.dev/guides/language/language-tour#final-and-const) | [finally](https://dart.dev/guides/language/language-tour#finally) | [operator](https://dart.dev/guides/language/language-tour#_operators) 2 | [void](https://medium.com/dartlang/dart-2-legacy-of-the-void-e7afb5f44df0) |
| [continue](https://dart.dev/guides/language/language-tour#break-and-continue) | [for](https://dart.dev/guides/language/language-tour#for-loops) | [part](https://dart.dev/guides/libraries/create-library-packages#organizing-a-library-package) 2 | [while](https://dart.dev/guides/language/language-tour#while-and-do-while) |
| [covariant](https://dart.dev/guides/language/sound-problems#the-covariant-keyword) 2 | [Function](https://dart.dev/guides/language/language-tour#functions) 2 | [rethrow](https://dart.dev/guides/language/language-tour#catch) | [with](https://dart.dev/guides/language/language-tour#adding-features-to-a-class-mixins) |
| [default](https://dart.dev/guides/language/language-tour#switch-and-case) | [get](https://dart.dev/guides/language/language-tour#getters-and-setters) 2 | [return](https://dart.dev/guides/language/language-tour#functions) | [yield](https://dart.dev/guides/language/language-tour#generators) 3 |
| [deferred](https://dart.dev/guides/language/language-tour#lazily-loading-a-library) 2 | [hide](https://dart.dev/guides/language/language-tour#importing-only-part-of-a-library) 1 | [set](https://dart.dev/guides/language/language-tour#getters-and-setters) 2 |  |
| [do](https://dart.dev/guides/language/language-tour#while-and-do-while) | [if](https://dart.dev/guides/language/language-tour#if-and-else) | [show](https://dart.dev/guides/language/language-tour#importing-only-part-of-a-library) 1 |  |
| [dynamic](https://dart.dev/guides/language/language-tour#important-concepts) 2 | [implements](https://dart.dev/guides/language/language-tour#implicit-interfaces) 2 | [static](https://dart.dev/guides/language/language-tour#class-variables-and-methods) 2 |  |
|  |  |  |  |

The following table lists the words that the Dart language treats specially.

Avoid using these words as identifiers. However, if necessary, the keywords marked with superscripts can be identifiers:

* Words with the **superscript 1** are contextual keywords, which have meaning only in specific places. They’re valid identifiers everywhere.
* Words with the **superscript 2** are built-in identifiers. To simplify the task of porting JavaScript code to Dart, these keywords are valid identifiers in most places, but they can’t be used as class or type names, or as import prefixes.
* Words with the **superscript 3**are limited reserved words related to [asynchrony support](https://dart.dev/guides/language/language-tour#asynchrony-support). You can’t use **await** or **yield**as an identifier in any function body marked with **async, async\***, **or sync\*.**

All other words in the table are reserved words, which can’t be identifiers.

**OPERATORS**:

| **Description** | **Operator** |
| --- | --- |
| unary postfix | *expr*++    *expr*--    ()    []    .    ?. |
| unary prefix | -*expr*    !*expr*    ~*expr*    ++*expr*    --*expr*      await *expr* |
| multiplicative | \*    /    %  ~/ |
| additive | +    - |
| shift | <<    >>    >>> |
| bitwise AND | & |
| bitwise XOR | ^ |
| bitwise OR | | |
| relational and type test | >=    >    <=    <    as    is    is! |
| equality | ==    != |
| logical AND | && |
| logical OR | || |
| if null | ?? |
| conditional | *expr1* ? *expr2* : *expr3* |
| cascade | .. |
| assignment | =    \*=    /=   +=   -=   &=   ^=   *etc.* |

1. **Tenary Operator:**

*Conditional*? *expr1* : *expr2*;

or

*Condition*?*expr1*??*expr1a* : *expr2*??*expr2a*.

1. **Null Aware Operator:**
2. **?.** called null-aware dot Operator :

class Num {

  int? num = 10;

}

main() {

  var n;

  //var n = Num();  This will help to print n.num, which is 10.

  int? number = n?.num;

  print(number);

}

**Explanation**:

*“****?.****” is used to check if the object “****n****” is* ***null****. If it is* ***null****, it will assign* ***null to number*** *but if it is* ***not null****, then the value of* ***num*** *in the object “n” is being assigned to* ***number*** *and the “****print(number);”*** *will print out the value, which in this case is 10.*

1. **null aware operator (??) :**

class Num {

  int? num;

}

main() {

  int? number;

  print(number?? 1);// this will print 1.

  print(number); // this will print null.

}

**Explanation**:

***??*** *was used to check if* ***number*** *is null, then it will assign it “1” else the value stored in* ***number*** *will be printed out.*

1. **??= (null aware operator) :**

main() {

  int? number;

  print(number ??= 1); // this will print 1.

  print(number); // this will print 1 also.

}

**Explanation**:

***??=*** *was used to check if* ***number*** *is null, then it will assign it “1” else the value stored in* ***number*** *will be printed out.*unlike **??**, if ‘**1’** is assigned to **number**, it will be **stored as the value** of **number** unless later changed.

**iv. .. & ..? (cascade & null aware cascade operators**):

Cascades (..) allow you to make a sequence of operations on the same object. In addition to function calls, you can also access fields on that same object. This often saves you the step of creating a temporary variable and allows you to write more fluid code.

Example:

void main(List<String> args) {

  List<int> l1 = [1, 0, 2];

  l1.sort();

  l1 = l1.reversed.toList();

  l1.addAll([5, 3, 4]);

  l1.sort();

  l1 = l1.map((e) => e + 1).toList();

  print(l1);

  //! Instead of the above we will use the below //cascade operator

  List<int>? l2 = [1, 0, 2]

    ..sort()

    ..reversed

    ..addAll([5, 3, 4])

    ..sort();

  print(l2.map((e) => e + 1)..toList());

}

*This will print* **[1,2,3,4,5,6]**

**(1,2,3,4,5, 6)**

**NB:**There is no comma sign after each called method or field and also that the “cascade operator” can be used to call a single method like in the “print(L2.map((e) => e + 1)..toList)”.

1. **… & …? (Spread & null aware spread operator):**

The “…” is used to add a list to another list with change in one not affecting the other one. The added list does not act as a single element in the one that you added it to but increases the new list that it is added to by it’s element number. like, if list1( having 3 elements) is added to list2( having 4 elements), then list2 will now have 7 elements.

The “…?” is used to add a list that can be null to another list, this will ensure that the list to be added is not null before adding it to the new list.

Example:

void main(List<String> args) {

  var a = [1, 2, 3], b, c = [7, 8, 9];

  var d = [...a, ...?b, ...c];

  // var e = [...a, ...b, ...c];

  print("a is $a");

  a = [3, 2, if(a.[2] == 3) 4 else 3];

  print("a is now $a but it won't affect 'd'.");

  print("d is $d\n and \n‘e’ is throwing and exception when i try to print it.");

}

**NB**: Notice that there is an ***if else*** statement in the List of the code above, Conditional statement is allowed in List.

1. **?[] :**

In **names?[0]**, **“?”** makes sure that **before** the element at **index “0**” of the list “**names**” is **assigned or used** that “**names**” is not null.

1. **Word Operators**:
2. **is** :

returns true if the object has the specified type.

void main(List<String> args) {

  int? x;

  if (x is int) {

    print(10.isEven);

  }

}

This will print nothing because **x** is null and null is no more a subtype of int.

1. **is! :**

returns true if the object doesn’t have the specified type.

1. **as**:
2. It is used in casting an object to a specific type.
3. It is also used to give a convenient name to an imported derivative.

Example:

import 'dart:math' as mat102;

1. **~ :**
2. Unary bitwise complement (0s become 1 s; 1 s become 0 s)
3. **!:**
4. it inverts the following expression when it **comes before** it. (changes **false** to **true**, and vice versa)
5. **(!)** is also called **null-assertion operator**. Here it **comes after** an expression or variable to tell dart that the variable or expression is not null.
6. The **==** operator tests whether two objects are equivalent. Two strings are equivalent if they contain the same sequence of code units( same words). To test whether two objects x and y represent the same thing, use the **==** operator. (In the rare case where you need to know whether two objects are the exact same object, use the [**identical()**](https://api.dart.dev/stable/dart-core/identical.html) function instead.)
7. **~/ :**

This is the **truncating division operator**, it removes all the remainder and print out only the **integer value**.

1. **Normal Operator**:

\*, /, -, +, %, <, >, &&, &, ||, |, etc

***Strictly speaking, the “double dot” notation for cascades isn’t an operator. It’s just part of the Dart syntax***

### ASSERT:

During development, use an **assert statement** — assert(condition, optionalMessage); — to disrupt normal execution if a boolean **condition** is false

The first argument to **assert** can be any expression that resolves to a **boolean value**. If the expression’s value is true, the assertion succeeds and execution continues. If it’s false, the assertion fails and an exception (an [AssertionError](https://api.dart.dev/stable/dart-core/AssertionError-class.html)) is thrown.

When exactly do assertions work? That depends on the tools and framework you’re using:

* Flutter enables assertions in [debug mode.](https://docs.flutter.dev/testing/debugging#debug-mode-assertions)
* Development-only tools such as [dartdevc](https://dart.dev/tools/dartdevc) typically enable assertions by default.
* Some tools, such as [dart run](https://dart.dev/tools/dart-run) and [dart2js](https://dart.dev/tools/dart2js) support assertions through a command-line flag: --enable-asserts.

In production code, assertions are ignored, and the arguments to assert aren’t evaluated.

1. .

**4.**

**LOOPS:**

1. **if and Ifelse:**

if (false) {

    print(10);

  } else {

    print(5);

  }

1. **Switch:**

Switch statements in Dart compare **integer**, **string**, or **compile-time constants** using **==.** The compared objects must all be **instances** of the **same class** (and not of any of its subtypes), and the class must not override ==. [**Enumerated types**](https://dart.dev/guides/language/language-tour#enumerated-types) work well in switch statements.

int? x = 1;

  n:

  switch (5) {

    case 1:

      print("one!");

      break;

    case 2:

      print("Two!");

      break;

    case 3:

      print("Three!");

      break;

    default:

      print("I know!");

      break n;

  }

1. **Break:**
2. **Normal break:**

Check the switch statement above, although break can be used in other loop like for loop, for-in loop, forEach loop, if and ifelse statement etc.

1. **Label break:**

Example:

Check the default statement in the above switch statement.

1. **Continue:**
2. **Normal continue:**

If(…)continue;

1. **Label break:**

case1:

…

continue label\_name;

1. **While and do\_While:**

while(condition){

…

}

**And**

do{

…

}while(condition);

1. **For**:
2. **Normal for:**

for(initialization; condition; control){

…

}

1. **For\_each:**

main(){

int num = [1, 2, 3];

num.forEach( (n) => print(n) );// when u use it with lambda expression.

**OR**

num.forEach(print); but this is better, infact instead of using the first eg that has an anonymous function use **for-in**, it will be faster

}

**Or**

main(){

var numbers =[1,2,3];

numbers.forEach(printNum);// Here a method is use instead of the Lambda //expression.

}

printNum(num){

print(num);

}

*From the above explanation, we can see that forEach loop takes a method as parameter.*

1. **For\_in:**

Int num = [1, 2, 3];

for( var n in num){

print(n);

}

This will print

1

2

3

**5.**

**COLLECTION:**

1. **ITERABLE:**
2. It is an abstract collection
3. It is traversed with the help of an iterator(curr, nextItem())
4. It’s length is infinite.
5. Don’t use **.length** to check if an iterable is empty because it can be painfully slow. Instead use **.isEmpty** or **.isNotEmpty field**.
6. You can’t use **“[]”** operator to access the elements of an Iterable object but you can read through the iterable by using **elementAt()**. This will loop till the index you want.

Example:

void main() {

  Iterable<String> iterable = const ['Salad', 'Popcorn', 'Toast'];

//String str = iterable[1]; is wrong, so use

// elementAt(1) instead.

  print(iterable.elementAt(1));//Popcorn

}

1. You can use Iterable with **for-in** as it is the **easiest way** to loop through Iterable.

Example:

void main() {

  Iterable<String> iterable = ['Salad', 'Popcorn', 'Toast'];

  for (final element in iterable) {

    print(element);

  }

}

1. You might want to check a condition that work for the whole Iterable, instead of using **for-in**, you can easily use **every(…) or any(…)**.

Example:

Use

return items.any((item) => item.containts(‘a’));

//if any single element satisfy the condition

return items.every((item) => item.length >= 5);

//if all the element satisfy the condition

Instead of

for (final item in items) {

if (item.length < 5) {

return false;

}

}

return true;

1. To access a particular element in an Iterable, you can use first **firstWhere()** method. **firstWhere()** take a **bool function as parameter**, in which if it results to true, it will print out the element it stopped at, else it will print what is in **orElse** optional parameter or **throw** a **StateError exception**.

Example:

void main() {

  Iterable<String> iterable = const ['Salad', 'Popcorn', 'Toast'];

  String popcorn = iterable.firstWhere((element) => element.contains("Pop"));

  print(popcorn);

}

**firstWhere()** also takes another function as an **optional named parameter** called **orElse**. **orElse** will be print out if the bool function didn’t later result to true else it will **throw StateError exception**.

1. When you want **only one** element of your Iterable to satisfy a particular condition used the **singleWhere(…)** method. It will throw an error when **more than one** element satisfy the condition or when **non** of the element satisfy the condition.

Example:

return items.singleWhere(

    (element) => element.startsWith('M') && element.contains('a'),

);

1. To return an Iterable that contain all the elements that satisfy a particular condition use **where(…)** but unlike **firstWhere** and **singleWhere**, **where(…)** don’t throw **StateError** exception.

Example:

var containsA = items.where(

  (a) => a.contains("a"));

print(containsA);

*This will print an Iterable that contains all the elements that have ‘a’ in it.*

1. To **modify the elements** of an Iterable before printing it out use **map(…)**. The example below shows an int Iterable made String Iterable with some additional string.

Example:

void main(List<String> args) {

  Iterable<int> nums = [1, -2, 3, 42];

  Iterable<String> numMadeStr = nums.map(

    (e) => "The no. is ${e.toString()}\n",

  );

  print(numMadeStr);

}

1. Trying to access an element of a **synchronous generator** will generate all element proceeding the desired element until the desired element is found.

Example:

void main(List<String> args) {

  var a = showGenerator(5);

  // print("a.first --> ${a.first}");

  // print("a.last --> ${a.last}");

}

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

The above code will print nothing on the screen.

void main(List<String> args) {

  var a = showGenerator(5);

  print("a.first --> ${a.first}");

  // print("a.last --> ${a.last}");

}

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

This will print:

*Generator start:*

*1*

*a.first --> 1*

void main(List<String> args) {

  var a = showGenerator(5);

  // print("a.first --> ${a.first}");

  print("a.last --> ${a.last}");

}

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

This will print:

*Generator start:*

*1*

*2*

*3*

*4*

*5*

*Generator End!*

*a.last --> 5*

void main(List<String> args) {

  var a = showGenerator(4);

  print("a.last --> ${a.last}");

  print("a.first --> ${a.first}");

}

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

This will print:

*Generator start:*

*1*

*2*

*3*

*4*

*Generator End!*

*a.last --> 4*

*Generator start:*

*1*

*a.first --> 1*

1. Notice the use of **yield** instead of **return** statement.
2. From the above four examples, what you want to print determines the output you will get.
3. **yield\*** is used to return an output of **Iterable<int>**, as **yield** returns an output of **int**.

Example:

void main(List<String> args) {

  var a = showGenerator(5);

  print("a.last --> ${a.last}");

  print("a.first --> ${a.first}");

}

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  yield\* \_showNegativeGenerator(n);

  print("Generator End!");

}

Iterable<int> \_showNegativeGenerator(int n) sync\* {

  print("Negative Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Negative Generator End!");

}

This will print:

*Generator start:*

*1*

*2*

*3*

*4*

*Negative Generator start:*

*1*

*2*

*3*

*4*

*Negative Generator End!*

*Generator End!*

*a.last --> 4*

*Generator start:*

*1*

*a.first --> 1*

1. **yield** and **yield\*** are used to hold the value of **int** and **Iterable<int>** in the above code respectively. They are just like memory that holds values until they are needed.
2. In the above Examples, **return** statement can’t be used because of **sycn\*** keyword.
3. If you create a class that can provide Iterators for use in **for-in loops**, extend (if possible) or implement Iterable. Implement Iterator to define the actual iteration ability.

**Example:**

class Process {

// Represents a process...

}

class ProcessIterator implements Iterator<Process> {

@override

Process get current => ...

@override

bool moveNext() => ...

}

// A mythical class that lets you iterate through all

// processes. Extends a subclass of [Iterable].

class Processes extends IterableBase<Process> {

@override

final Iterator<Process> iterator = ProcessIterator();

}

void main() {

// Iterable objects can be used with for-in.

for (final process in Processes()) {

// Do something with the process.

}

}

1. **.**

**VALUABLES:**

**any()**// Returns true if at least one element satisfies the condition eg.

*items.any((item) => item.contains('a'))*

**contains(…)//**Checks if the element contains the letter/RegExp and if the optional index is provided, it will start from that index to check.

**elementAt(index)**//Used to access element at the specified index as [] is not valid.

**every()**// Returns true if all elements satisfy the condition.

*items.every((item) => item.length >= 5)*

**first//** used to access first element of the iterable**.**

**firstWhere(…)//**Used to print out the element that satisfy a particular condition.

**.form(…)//** is used the form a list from an already existing iterable.

**isEmpty()//** Checks if the iterable is empty.

**isNotEmpty()//**

**last//** used to access last element of the iterable.

**singleWhere(…)//** This is just like firstWhere(…) but unlike firstWhere, it loops through the whole items to make sure that not more than 1 item satisfy the solution.

**skipWhile(…)//**This will start from where the element that satisfy a condition is reached.

**startWith(…)//**Checks if the element start with the letter/RegExp and if the optional index is provided, it will start from that index to check.

**takeWhile(…)//**This will print all the elements of an Iterable before a condition is satisfied( results to true).

**toList();//** is preferably used to preserve type than List.from(…). Although List<Type>.from(…) is more useful when you want to change the type.

**where(…)//** It returns an Iterable that contains all the element that satisfy a certain condition.

**whereType()//** This can be used instead of **where()** to filter a particular type from an iterable. like for **int**, use **iterable.whereType<int>();**

**NB: *takeWhile(..)*** *and* ***skipWhile(…)*** *are usually used to separate an Iterable into* ***two different*** *parts.*

1. **LIST:**
2. **It uses square bracket as below**

[element,…,element];

1. **It has many constructors, which are:**

[**List.empty**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.empty.html)**({**[**bool**](https://api.dart.dev/stable/2.17.1/dart-core/bool-class.html)**growable = false})//**Creates a new empty list.

[**List.filled**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.filled.html)**(**[**int**](https://api.dart.dev/stable/2.17.1/dart-core/int-class.html)**length, E fill, {**[**bool**](https://api.dart.dev/stable/2.17.1/dart-core/bool-class.html)**growable = false})//**Creates a list of the given length with fill at each position.

[**List.from**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.from.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.1/dart-core/Iterable-class.html)**elements, {**[**bool**](https://api.dart.dev/stable/2.17.1/dart-core/bool-class.html)**growable = true})//**Creates a list containing all elements.

[**List.generate**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.generate.html)**(**[**int**](https://api.dart.dev/stable/2.17.1/dart-core/int-class.html)**length, E generator(**[**int**](https://api.dart.dev/stable/2.17.1/dart-core/int-class.html)**index), {**[**bool**](https://api.dart.dev/stable/2.17.1/dart-core/bool-class.html)**growable = true})//**Generates a list of values.

[**List.of**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.of.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.1/dart-core/Iterable-class.html)**<E> elements, {**[**bool**](https://api.dart.dev/stable/2.17.1/dart-core/bool-class.html)**growable = true})//**Creates a list from elements.

[**List.unmodifiable**](https://api.dart.dev/stable/2.17.1/dart-core/List/List.unmodifiable.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.1/dart-core/Iterable-class.html)**elements)//**Creates an unmodifiable list containing all elements.

1. **List can be of fixed length or growable length.**

* **Fixed:**

  final fixedLengthList = List<int>.filled(5, 0); // Creates fixed-length list.

  print(fixedLengthList); // [0, 0, 0, 0, 0]

  fixedLengthList[0] = 87;

  fixedLengthList.setAll(1, [1, 2, 3]);

  print(fixedLengthList); // [87, 1, 2, 3, 0]

// Fixed length list length can't be changed or increased

  fixedLengthList.length = 0; // Throws exception

  fixedLengthList.add(499); // Throws exception

* **growable:**

  final growableList = <String>['A', 'B']; // Creates growable list.

  growableList[0] = 'G';

  print(growableList); // [G, B]

  growableList.add('X');

  growableList.addAll({'C', 'B'});

  print(growableList); // [G, B, X, C, B]

1. You can add a **comma** after the last item in a Dart collection literal. This trailing comma doesn’t affect the collection, but it can help prevent copy-paste errors.
2. **It can be assigned to another list so that any change in one don’t affect the other**

List n = [1, “Nnamdi”, true];

var x = […n];

n.forEach( (n) {

print(n);

});

1. **It can be duplicated so that any change in one will affect the other one**

List n = [1, “Nnamdi”, true];

var x = n; *// when u change 1 in n, x will also be affected*.

n.forEach( (n) {

print(n);

});

1. **It can have a specific type**

List<String> n = [“Hello”, “Nnamdi”, “true”];

n.forEach( (n) {

print(n);

});

1. **You can add data to a List through**

n[0] = 1;

or

n.add(“Okolo”);

1. **It uses the getter “length”.**

print(n.length);

**VALUABLES**:

**myList.add(“A”);**// used to add an element to the list.

**asMap(...);//**

**myList.clear();**// Used to clear the whole value of a List.

**myList.contains(“A”);**// This will print true b/c myList has “A”.

**myList.indexOf(“A”);//**Used to check the position of “A” in myList.

**myList.insert(2, “YES”);**// This is used to add “YES” to myList at index “2” without deleting the form element at index “2” but shift it to index “3”.

**myList.removeAt(0);**// Removes the element at index position “0”.

**myList.reversed**;

**myList.shuffle(…);**//

**myList.skip(2).toList();**// Used to skip the first 2 elements when printing the elements of myList and then converts it to List, else it will print it not as a List again.

**myList.sort(int Function(String, String)? compare)//**Sorts this list according to the **order specified** by the **[compare] function**. The [compare] function must act as a **[Comparator]**.

Example:

var numbers = ['two', 'three', 'four'];  
// Sort from shortest to longest.  
numbers.sort((a, b) => a.length.compareTo(b.length));  
print(numbers); // [two, four, three]

**myList.take(4).toList();**// Used to print the first 4 elements of the list and the “toList()” is used to print it out as a list.

1. **SET**:
2. A set in Dart is an unordered collection of unique items. Dart support for sets is provided by set literals and the [Set](https://api.dart.dev/stable/dart-core/Set-class.html) type.
3. **It uses curl braces**

var setName = <type>{…};

1. **When any of it’s elements is duplicated, the first one will override others**

var setName = {“Nnamdi”, “Okolo”, “Linus”, “Nnamdi”};

for(var x in setName) print(x);

or

setName.forEach(print);

***This will print***

*Nnamdi*

*Okolo*

*Linus*

The second Nnamdi will not be printed.

1. Set support **collection if** and **for**; that’s adding if statement and for loop in a set like in **list** and **map**.
2. Set also support **spread** and **null aware spread operators(… & …?)** like **list** and **map**.
3. Set **can’t** be accessed with “setName.[n]” like list b\c it does not have an order.
4. Set can be created from a List and other Set constructors, like:

set setList = set.from([1, 2, 3]);

[**Set**](https://api.dart.dev/stable/2.17.2/dart-core/Set/Set.html)**()//**Creates an empty [Set](https://api.dart.dev/stable/2.17.2/dart-core/Set-class.html).

[**Set.from**](https://api.dart.dev/stable/2.17.2/dart-core/Set/Set.from.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.2/dart-core/Iterable-class.html)**elements)//**Creates a [Set](https://api.dart.dev/stable/2.17.2/dart-core/Set-class.html) that contains all elements.

[**Set.identity**](https://api.dart.dev/stable/2.17.2/dart-core/Set/Set.identity.html)**()//**Creates an empty identity [Set](https://api.dart.dev/stable/2.17.2/dart-core/Set-class.html).

[**Set.of**](https://api.dart.dev/stable/2.17.2/dart-core/Set/Set.of.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.2/dart-core/Iterable-class.html)**<E> elements)//**Creates a [Set](https://api.dart.dev/stable/2.17.2/dart-core/Set-class.html) from elements.

[**Set.unmodifiable**](https://api.dart.dev/stable/2.17.2/dart-core/Set/Set.unmodifiable.html)**(**[**Iterable**](https://api.dart.dev/stable/2.17.2/dart-core/Iterable-class.html)**<E> elements)//**Creates an unmodifiable [Set](https://api.dart.dev/stable/2.17.2/dart-core/Set-class.html) from elements.

1. **Empty set is not define with empty curl braces unless an angular braces of the type of the empty set is used as in the below example.**

Var setName = <int?>{};

or

Set<int?> setName = {};

print(setName.runtimeType);

***This will print***

\_CompactLinkedHashSet<int?>

**VALUABLES**:

**mySet.add(“A”);//** used to add an element to the set.

**mySet,addAll(anotherSet);**// Used to add all the elements of another set.

**mySet.contains(String)**//

**mySet.containsAll(List)**//

**myList.remove(“A”);**// Removes the element “A” from the set.

**mySet.isEmpty();**// Used to check if the set is empty.

**mySet.from([1, 2.3, “Nnamdi”, true]);**// used to create set from a list.

1. **MAP**:
2. It uses curl braces like set but contains key/ value pair

var mapName = {

// key: value

“Name”: “Nnamdi”

“RegNo”: 2017234005

}

print(mapName[“RegNo”]);

***This will print***

*2017234005*

**Or**

**You can also use integer values as the key**

var mapName = {

1: “Nnamdi”,

2: “2017234005”

}

print(mapName[2]);

***This will print***

*2017234005*

1. Map also support ***spread operator*, *collection if*** and ***collection for*** like **Set** and **List** above.
2. Try to understand the below code on how to print keys/values.

void main(List<String> args) {

  Map<String, int> grades = Map();

  grades["pass"] = 50;

  grades['fail'] = 0;

  grades["perfect"] = 100;

  print("$grades\n");

  for (var item in grades.keys) {

    print(item);

  }

  print();

for (var item in grades.values) {

    print(item);

  }

}

1. **Empty map can be defined like**

var mNames = Map();

//then u can assign them values like

mNames[1] = “Nnamdi”;

mNames[“pass”] = 50;

**VALUABLES**:

**map.contains(Object)**// To retrieve all key or all value in a map

**map.containsKey(Object)**// To check if a map contain a Key.

**map.putIfAbsent(...)**// Used to assign a value to a map’s key. This is possible if and only if the key doesn’t exist earlier.

**map.keys**//

**map.remove(key)**// To remove a ley and it’s corresponding value.

1. **QUEUE:**

This is just like List but u need to import collection class. like;

import ‘dart:collection’;

//then

Queue queue = Queue();

queue.add(‘2’);

queue.removeFirst();

queue.removeLast();

queue.elementAt(0);

**6.**

**Sound Null-safety:**

* 1. **Soundness** is related with strength of Null-safety addition.
  2. **Null-safety** was added from dart 2.12, so setting your sdk version to be more than this version make your app **sound null-safe**.
  3. You can make a type **nullable** by proceeding it with **‘?’**, like

**int? num**;

* 1. if you don’t explicitly make a variable type null, it **can’t be null**. Thus there is no variable declaration like:

int num;//error

int num = 2;// good

int? num;// which is equivalent to the below but it’s better.

int? num = null;// manageable.

* 1. When you are sure that a **variable is not null** when you want to use it, **although you made it nullable** in declaration, use the **null assertion operator “!”** to tell dart the it won’t be null like the below example:

  int? couldBeNullButIsnt = 1; //declared nullable

  List<int?> listThatCouldHoldNulls = [2, null, 4];//Could //contain nullable int.

  int a = couldBeNullButIsnt;//

  int b = listThatCouldHoldNulls.first!; //Becase I know //that the first item in the list is non-null, I used the //null-assertion operator ‘!’ to access it.

  int c = couldReturnNullButDoesnt()!.abs(); // absolute //value

  print('a is $a.');

  print('b is $b.');

  print('c is $c.');

* 1. [**Type promotion**](https://dart.dev/null-safety/understanding-null-safety#type-promotion-on-null-checks) can be done **with exception** or **conditional statement**. The below example getLenght made String? str non- nullable. **Using a null-aware operator** on a variable inside a condition **doesn’t promote** the variable to a non-nullable type. If you want the variable **to be promoted** inside the body of the if statement, it might be better to use an explicit **“!= (or == or as)”** null check instead of **?.** followed by **??**. Caveat, **type promotion only works on local variables**, not on fields or top-level variables, so for you to make other non-local variable that are nullable non-nullable, you need to assign it to a nullable local variable and then use the nullable local variable in control flow.

Example:

String? name;

int getLength(String? str) {

name = str;

  if (str == null) { //name and str are promoted to non-//nullable types.

    throw ArgumentError(

      "You didn't assign any value to ",

    );

  }

  return str.length;

}

OR

int getLength(String? str) {

  if (str == null) {

    return 0;

  }

  return str.length;

}

void main() {

  print(getLength('This is a string!'));

}

* 1. When you put **late** in front of a variable declaration, that tells Dart the following:
* Don’t assign that variable a value yet.
* You *will* assign it a value later.
* You’ll make sure that the variable has a value *before* the variable is used.

So, **late** can be used to declare a non-nullable field without assigning it a value. example:

late int num;

late final num;

late final int num;

1. To check that you have Dart 2.12 or later use the below command in your cmd

.../dart --version

1. To get the migration state of your package’s dependencies, using the following command in your cmd.

.../dart pub outdated --mode**=**null-safety

1. Run …/dart pub upgrade --null-safety to upgrade to the latest versions supporting null safety. **Note:** This command changes your pubspec.yaml file.
2. Dart also ***removed implicit downcast***(Object can be passed to String without explicitly adding ***as String*** to it) with the invention of null-safety, so if you want to pass a super class to any of it’s subclass you must explicitly add ***as subclass***. This is because a nullable type(like String?) is a superclass of non-nullable type(String), thus this is to ensure that null is not assigned where it shouldn’t be.
3. **With null safety.**

* **Top level variable and static field declarations must have an initialize or they will be declared as nullable.**
* **Instance fields must either have an initializer at the declaration, use an initializing formal, or be initialized in the constructor’s initialization list(before the constructor’s body) or they will be declared as nullable.**
* **A local variable must be definitely assigned before it is used but it can be declared without initialization.**
* **Optional and named parameters must have a default value or they will be declared as nullable.**
* ***final local variable* can be declared and initializes in control flow statement.**
* **Normally instance field can’t access this keyword** because you don’t have access to the new object until all field initializers have completed. But with a late field, that’s no longer true, so you can access **this**, **call methods**, or **access fields** on the instance.
* ***late final fields*** unlike ***normal final******fields***, you do not have to initialize the field in its declaration or in the constructor initialization list. You can assign to it later at runtime. But you can only assign to it once and that fact is checked at runtime.
* **Abstract field that is non-nullable can be declared in abstract classes.**

1. **.**

**7.**

**FUNCTION**

1. **Function:**
2. This is a type in Dart programming language; it is an abstract class.
3. It is used to receive a method as a variable.

Example:

int mNum() {

  return 5;

}

void main(List<String> args) {

  Function one = mNum;// You can not assign “mNum()” //to one, it will result to error

  print(one);// This will print mNum reference

//Closure: () => int from Function 'mNum': static.

  print(one());//This will print "5". So “one()” //now works like “mNum()” and one like mNum.

}

1. In the **example above**, you can replace “**Function one = mNum**;” with “**var one = mNum**;”. Then when you hover over “**one**” you will see “**int Function() one**”. This shows that **one** is another name for **mNum** and that **mNum** has a return type of **int**.

**NB**: *Please try using* ***var one = mNum*** *to see what I meant.*

Example:

int mNum(int Function(int j) e) {

  return 7 + e.call(7);

  // OR

  // return 7 + e(7);

  //will also work very fine.

}

int c(n){

if (n is int) {

    return n + 6;

  } else {

    return 6;

  }

}

void main(List<String> args) {

  var one = mNum;

  print(one); // This will print Closure:...etc.

  print(one(c)); //This will print "20"

}

Explanation:

\_\_**print(one)** will print the below.

Closure: ((int) => int) => int from Function 'mNum': static.

\_\_**print(one(c))** will print the value **20** because.

1. When **one(c)** is being called, it will return **7+e(7)** .
2. Then 7 trying to sum up with an int, will make e(7) *(which is a function and the parameter of* ***one(c)***) to be called.
3. **e(7)** is being represented by the function and parameter “**e”**, which is represented by the argument “**c”** (which is a function).
4. Then **c(7)** is called,
5. **Now explain the below codes.**

int mNum(int Function(int j) e, int a) {

  return e(a);

  // OR

  // return 7 + e(7);

  //will also work very fine.

}

int c(n) {

  return n;

}

void main(List<String> args) {

  var one = mNum;

  // You can not assign mNum(), it will result to error

  print(one); // This will print mNum reference

  print(one(c, 9)); //This will print "9"

}

**Do you see how you use a single method call to return another method with it’s parameter from the above code.**

String Function(int) makeCar(String make) {

  var engine = "4.4 V8";

  return (model) => "$make $model $engine";

}

void main(List<String> args) {

  var a = makeCar("BMW");

  print(makeCar("BMW"));

or

print(a); // This will return information of the function it is meant to return, which is a Function with parameter of type **int** and a return type of **String**.

  print(a(7)); // This prints the real output needed, which is

  // BMW 7 4.4 V8.

  //This is because "a" represents the return type of makeCar(...).

}

**Try to Understand the above code.**

1. **.**
2. You can also **call an instance of a Dart class** as a function with **ClassName()()** when you must have **implemented call() method** in the **class**.
3. Dart **supports top-level functions** (such as main()), as well as **functions tied to a class or object** (static and instance methods, respectively). You can also create **functions within functions** (nested or local functions).
4. **Instance**, **getter**, and **setter methods** **can be abstract**, defining an interface but leaving its implementation up to other classes.
5. Only an **expression**—**not a *statement***—can appear between the arrow (=>) and the semicolon (;). For example, you **can’t** put an [**if statement**](https://dart.dev/guides/language/language-tour#if-and-else) there, but you can use a [**conditional expression**](https://dart.dev/guides/language/language-tour#conditional-expressions)**.**
6. **Anonymous function:**

An anonymous function looks similar to a named function— **zero or more parameters**, separated by commas and optional type annotations, between parentheses.

1. **Instance methods** on objects **can access** **instance variables** and **this** keyword.
2. You can also assign a function to a variable, such as:

Example:

  var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';

  assert(loudify('hello') == '!!! HELLO !!!');

1. **Functions can be in the form of :**
2. **Arrow Function:**

int mName(int n) => n\*n;

notice that there is no **curl braces**({}), nor **return statement**(return type) and it consist of **one expression**.

1. **Anonymous/Lambda Function:**

This can be assigned to a **variable** Like below:

var aMeth = (){

...

}

Or **as a parameter**( as in **forEach** loop).

1. **Normal Function:**

int mName(int n){

…

return n\*n;

}

1. You **can’t invoke** an unimplemented method unless **one** of the following is true:

* The receiver has the static type dynamic.
* The receiver has a static type that defines the unimplemented method (abstract is OK), and the dynamic type of the receiver has an implementation of noSuchMethod() that’s different from the one in class Object.

1. .

**7a.**

**OVERRIDING**

1. Method/Function overriding is **replacing the functionality** of a method **in a subclass**.
2. Optionally, you can use the **@override** annotation to annotate the method but as I said it is optional. please try adding it, it will help you not to make mistakes.
3. An overriding method declaration must match the method(s) that it override(s) in several ways:

* The **return type** must be the **same type** as (or a **subtype** of) the overridden method’s return type.
* **Argument types** must be the **same type** as (or a **supertype** of) the overridden method’s argument types. In the preceding example, the contrast setter of SmartTelevision changes the argument type from int to a supertype, num.
* If the overridden method accepts n positional parameters, then the overriding method must also accept n positional parameters.
* A [**generic method**](https://dart.dev/guides/language/language-tour#using-generic-methods) **can’t** **override** a **non-generic method**, and a **non-generic method** can’t override a **generic method.**

1. when you want to override a method but you want to replace **a type** of it’s parameter with it’s **subclass type,** use the **covariant** keyword as used in the **Cat class chase’s method parameter** below.

Example:

class Animal {

  final String name;

  Animal({required this.name});

  void whatAmI() => print("I'm an Animal.");

  void chase(Animal a) {}

}

class Mouse extends Animal {

  Mouse() : super(name: "Jerry");

}

class Cat extends Animal {

  Cat() : super(name: "Tom");

  @override

  void chase(covariant Mouse a) {}// here convariant is used to tighten the subclass //to it’s super class.

}

1. **Subclasses** can override **instance methods**( including [**operators**](https://dart.dev/guides/language/language-tour#_operators)), **getters**, and **setters**.
2. .

**8.**

**OPERATOR AND TYPEDEFS**

**Operators**

Operators are instance methods with special names. Dart allows you to define operators with the following names:

|  |  |  |  |
| --- | --- | --- | --- |
| **<** | **+** | **|** | [] |
| **>** | **/** | **^** | []= |
| **<=** | **~/** | **&** | ~ |
| **>=** | **\*** | **<<** | == |
| **–** | **%** | **>>** |  |

**Note:** You may have noticed that some [operators](https://dart.dev/guides/language/language-tour#operators), like **!=**, are not in the list of names. That’s because they’re just syntactic sugar. For example, the expression e1 != e2 is syntactic sugar for !(e1 == e2).

An operator declaration is identified using the built-in identifier operator. The following example defines **Adder** addition (+). Try and re-write the below code, it will help you to understand operator. Using operator is actually very easy.

class Adder {

  final int a, b;

  const Adder(this.a, this.b);

  String operator +(Adder adder) => "${a + adder.a} and ${b + adder.b}";

}

void main() {

  var obj = Adder(2, 5);

  var obt = Adder(3, 4);

  String output = obj + obt;

  print(output);

}

To test whether two objects x and y represent the same thing, use the == operator. (In the rare case where you need to know whether two objects are the exact same object, use the [identical()](https://api.dart.dev/stable/dart-core/identical.html) function instead.) Here’s how the == operator works

## Typedefs:

A type **alias** — often called a typedef because it’s declared with the keyword**typedef** — is a concise way to refer to a type. Here’s an example of declaring and using a type alias named IntList:

typedef IntList = List<int>;

IntList il = [1, 2, 3];

A type alias can have type parameters:

typedef ListMapper<X> = Map<X, List<X>>;

Map<String, List<String>> m1 = {}; // Verbose.

ListMapper<String> m2 = {}; // Same thing but shorter and clearer.

**Version note:** Before 2.13, typedefs were restricted to function types. Using the new typedefs requires a [**language version**](https://dart.dev/guides/language/evolution#language-versioning)of at least 2.13.

We recommend using [**inline function types**](https://dart.dev/guides/language/effective-dart/design#prefer-inline-function-types-over-typedefs)instead of typedefs for functions, in most situations. However, function typedefs can still be useful:

typedef Compare<T> = int Function(T a, T b);

int sort(int a, int b) => a - b;

void main() {

  assert(sort is Compare<int>); // True!

}

**9.**

**PARAMETERS**

**We** **have** :

1. **Required Positional parameter:**

It is being received by positional argument. This is the normal argument and parameter you know from java.

requiredPositional(int a, int b) {

  //? int a and int b were allowed without adding “?” because the compiler knows that you

  //? must assign it when the function is being called.

  print((a = 5) + b);

}

void main(List<String> args) {

  requiredPositional(10, 2);

}

NB: Notice that there is no required keyword in the parameter above, If you include it, it will throw an error.

1. **Optional-Positional parameter:**
2. These are positional parameter that can be skipped when calling the method that have them as parameter.
3. They are placed inside a square bracket.
4. It placed after all the **required position parameters**
5. It should be made nullable or given a default value.

Example:

optionalPositional([int? a, int b = 7]) {

  //? "int b"can not be in the above parameter unless //it will throw error!

  print((a = 5) + b);

}

void main(List<String> args) {

  optionalPositional(

    8,

    5,

  );

}

**NB**: : *You* ***can’t have*** *both* ***Optional positional parameter*** *and* ***Optional-Named parameter*** *in the same argument list.*

1. **Optional-Named parameter:**
2. It is being enclosed with curl brace ”{}”
3. There is no restriction to the order of calling it. You must call them with their names.

optionalNamed({int? a, int b = 7}) {

  //? "int d"can not be in the above parameter unless it will throw error!

  print((a = 5) + b);

}

void main(List<String> args) {

  optionalNamed(

    a: 8,

    b: 9,

  );

}

// notice that the parameters passed is not complete

// there is a colon sign

//the other changed, which is by choice.

1. It must be placed after all the **required position parameters**
2. It should be **annotated** to be **required** or **made nullable** or **given a default value**
3. .

**NB**: *You* ***can’t have*** *both* ***Optional positional parameter*** *and* ***Optional-Named parameter*** *in the same argument list.*

1. **Required Named parameter:**
2. This is a Named parameter that is annotated to be required, thus, it must be assigned are function call.
3. .

requiredNamed({required int a, required int b}) {

  //? "int d"can not be in the above parameter unless it will throw error!

  print((a = 5) + b);

}

void main(List<String> args) {

  requiredNamed(

    a: 8,

    b: 9,

  );

}

1. You can also pass lists or maps as default values.

**Example:**

void doStuff(

    {List<int> list = const [1, 2, 3],

    Map<String, String> gifts = const {

      'first': 'paper',

      'second': 'cotton',

      'third': 'leather'

    }}) {

  print('list:  $list');

  print('gifts: $gifts');

}

1. .

**NB: we only have *positional* and *named parameter* in dart. Others are auxiliary parameters. positional arguments must come first before named arguments when u want to combine them.**

**10.**

**CONSTRUCTORS**

1. **Constructor** is a method that has the **same name as** that of the **class name**. It does not have return type. Although when the **keyword factory** proceed it, it will require a return statement as you will soon see later below.

example:

ClassName(String name, int age){

……

}//full basic declaration of a constructor.

Or

ClassName(this.name, this.age);// *this is used when there is already a variable with the name “name” and “age”*.

1. **final variable** can’t be initialized inside a constructor body.
2. **Non-Late or Non-Nullable final** **variables** are not initialized inside a constructor body but it can be b4 the constructor body like:

class A{

final int a;

A():a = 10, super(){

print(“You can remove the constructor body entirely and” “terminate with semi-colon”);

}

}

1. we also have a **named constructor**. Use a named constructor to implement multiple constructors for a class or to provide extra clarity.

Example:

ClassName.anyName(this.name, [this.age = 24]);

it’s object can be created like:

var obj = ClassName.anyName(“Nnamdi”, 25);

1. **factory Constructor:**

Use the **factory keyword** when implementing a constructor that doesn’t always create a new instance of its class. For example, a factory constructor might return an instance from a cache, or it **might** **return an instance of a subtype**. Another use case for factory constructors is initializing a final variable using logic that can’t be handled in the initializer list.

Example:

import 'dart:math';

class Point {

  final int x, y;

  const Point({required this.x, required this.y});

  factory Point.random({required bool isPositive}) {

    int minNegativeValue = -99,

        maxNegativeValue = -1,

        minPositiveValue = 0,

        maxPositiveValue = 99;

    int randomNegValue = minNegativeValue +

            Random().nextInt(

              maxNegativeValue - minNegativeValue,

            ),

        randomPosValue = minPositiveValue +

            Random().nextInt(

              maxPositiveValue - minPositiveValue,

            );

    return isPositive

        ? Point(

            x: randomPosValue,

            y: randomNegValue,

          )

        : Point(

            x: randomNegValue,

            y: randomPosValue,

          );

  }

  static const Point origin = Point(

    x: 0,

    y: 0,

  );

  @override

  String toString() => "Point(x: $x, y: $y)";

}

void main(List<String> args) {

  var trial = Point.random(isPositive: true),

      tester = Point.random(isPositive: false);

  print(trial);

  print(tester);

}

1. You can have **many constructor** with **factory keyword** in one class.
2. **factory constructor** returns an object of it’s class type.

like the above constructor can **return origin** instead of the const constructor it returned.

1. **factory** constructor does not use **this** keyword.
2. Invoke a factory constructor just like you would any other constructor:
3. DONE!.
4. **Constructor aren’t inherited**, which means that a superclass’s constructor is not inherited by a subclass. If you want a subclass to be created with a constructor defined in the superclass, you must implement that constructor in the subclass.
5. If the superclass doesn’t have an unnamed, default(no-argument) constructor, then **you** **must manually call one of the constructors in the superclass**. Specify the superclass constructor after a colon (:), just before the constructor body (if any).
6. **Redirecting Constructor**:

Sometimes a constructor’s only purpose is to redirect you to another constructor in the same class. A redirecting constructor’s body is empty, with the constructor call appearing after a colon (:).

class Point {

  double x, y;

  // The main constructor for this class.

  Point(this.x, this.y);

  // Delegates to the main constructor.

  Point.alongXAxis(double x) : this(x, 0);

}

1. **Constant Constructor**:

If your class produces objects that never change, you can make these objects compile-time constants. To do this, define a **const constructor** and make sure that **all instance variables** are **final**.

class ImmutablePoint {

  static const ImmutablePoint origin = ImmutablePoint(0, 0);

  final double x, y;

  const ImmutablePoint(this.x, this.y);

}

1. **Arguments to the superclass constructor** don’t have access to **this**. For example, arguments can call static methods but not instance methods.
2. The right-hand side of an initialize list **doesn’t** have access to **this**. During development, you can validate inputs by using **assert** in the initializer list.

Example:

class A {

  final x;

  final str = '#0';

  A.withAssert(this.x, this.y) : assert(x >= 0) {

    print('In Point.withAssert(): ($x, $y)');

  }

}

1. DONE!.

**11.**

**ENUM**

1. It is declared as below:

**enum** condition{

sunny,

cloudy,

drizzly,

rainy

}

1. It can be used like below.

enum condition { sunny, cloudy, drizzly, rainy }

void main(List<String> args) {

  var cond = condition.sunny;

  switch (cond) {

    case condition.cloudy:

      print("it' cloudy!");

      break;

    case condition.sunny:

      print("It's sunny!");

      break;

    case condition.drizzly:

      print("It's drizzly raining.");

      break;

    case condition.rainy:

      print("It's Raining!");

      break;

    default:

      print("Nothing");

      break;

  }

}

When you don’t use all the constant from your enum in the switch cases, you will get a warning message.

1. Enum has a **zero-based index** pattern.
2. Each value in an **enum** has an **index** **getter**, which returns the position of the value in the **enum** declaration. For example, the first value has index **“0”**, and the second value has index **“1”**.

Example:

assert(Color.red.index == 0);

assert(Color.green.index == 1);

assert(Color.blue.index == 2);

1. To get a list of all of the values in the **enum**, use the enum’s  **values**  constant field.

Example:

List<Color> colors = Color.values;

assert(colors[2] == Color.blue);

1. Enumerated types have the following limits:
2. You can’t **subclass**, **mixin**, or **implement** an enum.
3. You **can’t explicitly instantiate** an **enum**.
4. **.**

**12.**

**CLASESES AND INTERFACE**

1. **Normal Class:**

It does not have abstract method.

1. **Abstract Class:**
2. it uses the keyword “abstract”.
3. it can not be instantiated
4. it has abstract methods that must be implemented by all non-abstract class that extends it; abstract methods are method that don’t have body.
5. It can be declared and used in polymorphism.
6. Only abstract classes have abstract methods but if an **abstract method** is to be declared **in a normal class**, the method should be proceeded with the **external** keyword.

Example:

abstract class AClass {

  num meth();

  void nmeth() {

    print("AClass");

  }

}

class C extends AClass {

external int x();

  @override

  num meth() {

    return 5;

  }

}

//Interface

class B implements AClass {

  @override

  num meth() {

    // TODO: implement meth

    throw UnimplementedError();

  }

  @override

  void nmeth() {

    // TODO: implement nmeth

  }

}

1. To allow an instance of your Dart class to be called like a function, implement the **call()** method.
2. Try and study what the below example have.

Example:

class A {

  String str = strMth();

  String get strGet => str;

}

String strMth() {

  print("I am just a string but Nnamdi\n"

      "want to know if i will print out b4\n"

      "my official call.");

  return 'Done!';

}

void main(List<String> args) {

  var strM = A();

  print('\n\n');

  print(strM.strGet);

}

After running the program,try to include **late** **before str** in class A. late make the initialization of a variable to be done when you want to use it and not when it is declared.

1. [**Extension methods**](https://dart.dev/guides/language/language-tour#extension-methods)are a way to add functionality to a class without changing the class or creating a subclass.

Example:

extension A on int {

  int get by2 => this \* 2;

}

void main(List<String> args) {

  print(10.by2);

}

1. .

**INTERFACE**

1. Dart has **no interface keyword**. Instead, all classes implicitly define an interface. Therefore, you can implement any class.
2. Interfaces are classes that are implemented instead of extending them. Check the above example.
3. All methods **including normal methods** in any class that was implemented must be overridden.
4. **comma** is used to separate more than one interfaces that are implemented.
5. Every class implicitly defines an interface containing all the instance members of the class and of any interfaces it implements. If you want to create a class A that supports class B’s API without inheriting B’s implementation, class A should implement the B as interface.

**13.**

**“this” AND “super” KEYWORD**

**“this” Keyword:**

* 1. The **this.** syntax before a constructor parameter is called an “**initializing** **formal**”.
  2. It can be used to shorten argument name. eg:

class ClassName{

String name;

int age;

ClassName(this.name, [this.age = 12]) ;

//instead of

ClassName(String name, [int age = 12]);

}

* 1. The right-hand side of an initializer doesn’t have access to **this**. During development, you can validate inputs by using **assert** in the initializer list, as in the example below.

Example:

  A.withAssert(this.x, this.y) : assert(x >= 0) {

    print('In Point.withAssert(): ($x, $y)');

  }

**“super” KEYWORD:**

1. It is used to refer to the parent class.
2. Super class constructor is implemented before the constructor body like below.

eg:

ChildClass():super(); //this is for empty constructor.

ChildClass.name() : super.sName(){

…..

}// when u use both named child constructor and named super constructor.

class Vehicle {

  String model;

  int year;

  Vehicle(this.model, this.year);

  void showOutput() {

    print(model);

    print(year);

  }

}

class Car extends Vehicle {

  double price;

  Car.accord(String model, int year, this.price) : super(model, year);

  @override

  void showOutput() {

    super.showOutput();

    print(price);

  }

}

main(List<String> args) {

  var acc = Car.accord("Honda Accord", 2021, 15000.00);

  acc.showOutput();

}

**15.**

**GETTERS AND SETTERS**

1. Getter needs return type but setters don’t. check the program below.
2. Getters don’t need parameter nor argument but Setters need exactly one argument.

Example

class Meal {

  late String \_description;

  set description(String desc) {

    \_description = 'Meal description: $desc';

  }

  String get description => \_description;

}

void main() {

  final myMeal = Meal();

  myMeal.description = 'Feijoada!';

  print(myMeal.description);

}

1. All instance variable has an implicit getter, plus a setter if appropriate. You can create additional properties by implementing getters and setters, using the **get** and **set** keywords:
2. Getter and Setter are usual with the same name.
3. DONE!.

**16.**

**EXCEPTION HANDLING**

1. You can define a custom exception by implementing the Exception interface:
2. **throw** is used to throw an exception

Example:

throw FormatException(“Cause of the Exception.”);

1. **rethrow** is used to partially handle an exception, while allowing it to propagate. If you decide to rethrow an exception, prefer using the rethrow statement instead of throwing the same exception object using throw. **rethrow preserves** the original stack trace of the exception. **throw** on the other hand **resets** the stack trace to the last thrown position.
2. **on keyword:**

A **catch clause** with no ‘**on qualifier’** catches anything thrown by the code in the try block, which is not the best, thus try to use the ‘**on keyword’** to filter out the particular exception.

1. **try block** is used to accommodate the code that will throw the expection.
2. **catch** is used after the try block to catch the exception as shown below
3. **finally** is used to contain those statement that must be executed whether the exception is thrown or not.
4. You can also use the key word “**on**” when the exception to be thrown is know earlier.Eg:

try{

….

}on OutOfBoundException{

….

}on Exception catch(e){

…..

}catch(e){

…..

}finally{

…..

}

1. You can specify one or two parameters to **catch()**. The first is the exception that was thrown, and the second is the stack trace (a [StackTrace](https://api.dart.dev/stable/dart-core/StackTrace-class.html) object).

  try {

    // •••

  }

catch (e, s) {

    print('Exception details:\n $e');

    print('Stack trace:\n $s');

  }

1. To partially handle an exception, while allowing it to propagate, use the  **rethrow** keyword; like You want the cause of the exception to be known and still don’t want it to be handled unless the user correct the mistake. It can also be used for security checks – by scanning the code to see what wrong – before telling the user that there is an error.

  try {

    dynamic foo = true;

    print(foo++); // Runtime error

  } catch (e) {

    print('misbehave() partially handled ${e.runtimeType}.');

    rethrow; // Allow callers to see the exception.

  }

1. DONE!

**17.**

**GENERICS**

1. **Generic methods:**

notice where the angular bracket is placed.

M mine<M>(List<M> list) {

  M tmp = list[0];

  return tmp;

}

1. **Generic classes:**

class MyC<T extends num> {

  T a;

  T b;

  T add() {

    return (a + b);

  }

}

1. Try and run the below code.

class Stack<T> {

  final List<T> \_stack = [];

  T? get peak => \_stack.isNotEmpty ? \_stack.last : null;

  int get length => \_stack.length;

  bool get canPop => \_stack.isNotEmpty;

  T pop() {

    final T last = \_stack.last;

    if (\_stack.isNotEmpty) \_stack.removeLast();

    return last;

  }

  push(T value) => \_stack.add(value);

}

void main(List<String> args) {

  var stackInt = Stack<int>();

  var stackString = Stack<String>();

  stackInt.push(2);

  stackInt.push(3);

  stackInt.push(1);

  stackInt.push(5);

  stackString.push("Okolo");

  stackString.push("Linus");

  stackString.push("Nnamdi");

  stackString.push("Alphonsus");

  print(stackString.peak);

  print(stackString.length);

  print(stackString.pop());

  print(stackString.length);

  print(stackInt.peak);

  print(stackInt.length);

  print(stackInt.pop());

  print(stackInt.length);

}

1. Generics is also used to avoid code duplication. Generics let you share a single interface and implementation between many types, while still taking advantage of static analysis.
2. List, set, and map literals can be parameterized. Parameterized literals are just like the literals you’ve already seen, except that you add <type> (for lists and sets) or <keyType, valueType> (for maps) before the opening bracket.
3. Restricting a generic (parameterized) type, you might want to limit the types that can be provided as arguments, so that the argument must be a subtype of a particular type. You can do this using **extends**.
4. .

**18.**

**MIXINS**

1. This is the use of “**with**” keyword to share from the functionality of a class.

class Animal {

  final \_name = "Animals";

  Animal() {

    //name = "Animal";

    // ignore: unnecessary\_this

    print("$\_name  default constructor.");

  }

  Animal.myBaseClass() {

    print("$\_name named constructor.");

  }

  void name() => print(\_name);

  void move() => print("$\_name is walking.");

}

class Dog extends Animal with Cat {

  @override

  // ignore: overridden\_fields

  final \_name = "Dog";

  Dog() : super.myBaseClass() {

    print("$\_name defult constructor with named super contructor.");

  }

  Dog.myDogClass() {

    print("$\_name named constructor with super default constructor");

  }//Notice that this implements super class default

  //constructor by default.

}

class Cat {

  catAndDog() {

    print("Cat and Dog is used to indicated 2 enemies.");

  }

}

main(List<String> args) {

  Animal animal = Animal();

  animal.name();

  Animal poly = Dog.myDogClass();

  poly.name();

  //poly.catAndDog();//this shows error becauce poly is an Animal object.

  Dog dog = Dog();

  dog.catAndDog(); // This works perfectly;

}

1. **Mixin** is a class without constructor so it can’t be instantiated.
2. A class can share from **more than one mixin** at a time and you don’t need to re-implement it’s whole fields like you do in **interface**.
3. **Mixin** can also have abstract and normal methods like abstract class.
4. The only difference between a **Mixin** and an **abtract class** is that, it does not extend any class.
5. Both **abstract classes** and **normal classes** can serve as a **mixin** but remember, they are not meant to extend any class other than **Object** and they **should not contain any constructor**.
6. You can also specify those class that should be mixed with a **mixin** by using the **on** keyword, when you try to mix a mixin with a class without extending/implementing the class that was specified to be used with that mixin, your code will not compile.

Example:

class A {

  void p() => print("A class.");

}

mixin B on A {

  void pl() => print("B mixin");

  @override

  void p() => pl();

}

mixin C {

  void pl() => print("C mixin");

  void p() => pl();

}

class D extends A with B, C {}

void main(List<String> args) {

  D d = D();

  d.p();

}

if class D did not extend the class A, it can not mix with the mixin B.

1. When you run the above code, it will print out ‘**C mixin’** because **C** is the **last mixin** stated.
2. DONE!

**19.**

**EXTENSION**

* 1. [Extension methods](https://dart.dev/guides/language/language-tour#extension-methods) are a way to add functionality to a class without changing the class or creating a subclass. This is a keyword used to extend classes that can’t be extended like **int**, **String**… types.

Example:

extension A on int {

  int get n => this \* 2;

}

void main(List<String> args) {

  print(10.n);

}

* 1. You can’t invoke extension methods on variables of type **dynamic**. For example, the following code results in a runtime exception:

extension Mine on String {

  static dynamic \_n = "3";//Change the dynamic to String for it to pass

  int toInt() {

    return int.parse(this);

  }

  static int get n => \_n.toInt();

}

void main(List<String> args) {

  print(Mine.n);

}

* 1. extension can’t declare instance field that is why the above examples field is static. it can be static final like below.

static final String \_n = "3";

* 1. To create a local extension that’s visible only in the library where it’s declared, either omit the extension name or give it a name that starts with an underscore (\_).
  2. The members of the extension can be **methods**, **getters**, **setters**, **operators**. Extensions can also have **static fields** and **static helper methods** etc.
  3. .

**20.**

**PACKAGES**

1. **Funtionalities:**

* **pubspec.yaml:**
* It is used for version control.
* When you want to create a package manually, create a folder and then add pubspec.yaml file to it. In the pubspec.yaml file add name and environment field, then run **> dart pub get** on your terminate, this will create other packages like **.packages, analysis\_options.yaml, pubspec.lock and dart\_tool** folders.

*Remember that the ending folder in the terminal should be in the name of the dart package you want to create, like:*

***c:\src\your\_new\_package\_name***

* In VScode, when you **save pubspec.yaml** file, the file will run making sure that all functionalities in the file is activated properly. These functionalities includes: **dependencies**, **environment**, **dev\_dependencies** etc. You can use **> dart pub get** to run this also.
* **pubspec.lock:**

It contains the list of all packages your package depends on but it is deprecated now.

* **package\_config.json:**

It contains the list of all packages your package depends on.

* **analysis\_options.yaml:**

It is the list of rules that the dart analyzer uses in scanning your code. This rule and called **lint rule**.

* **.**

1. **Uses:**

* **Http**:

It is used to retrieve data from an API.

* **Json\_serializable:**

Used to pass a json to a specific class.

* **Hive**:

Used to store data to a **local and fast database**.

* .

**LET’S CREATE A PACKAGE, SOME FILES INSIDE IT AND THEN USES THEM:**

1. This is how to create a sub\_package inside our main package and using their files.
2. Assuming your main package name is “**mine**”.
3. Inside **mine** you can create **packages** folder.
4. inside **packages**, let’s create two packages(folders) called **home/lib** and **school/lib**.
5. inside **home’s** **lib** folder create **src/address.dart**, **src/parents.dart** and **src/siblings.dart** dart files like:

**> cd home/lib/src/address.dart**

1. Do no.5 for **school** with **src/reg\_no.dart**, **src/level.dart** and **src/department.dart** like:

**> cd school/lib/src/level.dart**

1. Now create a folder called output on both lib folder, like:

**> cd home/lib/output .dart**

**and**

**> cd school/lib/output.dart**

1. implement a method in each of **address**, **parents**, **siblings**, **reg\_no**, **level** and **department** dart files.
2. Now **export** them to their respective **output** dart file so that you will only import their **output** file when you want to use them.

For Home/lib/output:

export 'src/address.dart';

export 'src/parents.dart';

export 'src/siblings.dart';

For School/lib/output:

export 'src/reg\_no.dart';

export 'src/level.dart';

export 'src/department.dart';

From the above two samples, only the output file is needed to be imported for other files

1. Now to make **home** and **school** a package, let’s create the file **pubspec.yaml** in each of them. Then add **name** and **environment** fields to each of the **pubspec.yaml** file. example:

name: home

environment:

  sdk: '^2.14.0'

1. Also add **analysis\_options.yaml** file to **home** and **school**. This will indicate a warning until u add **lints** to your **pubspec.yaml** file for each of the two packages.

example:

name: school

environment:

  sdk: '>=2.14.4 <3.0.0'

dev\_dependencies:

  lints: ^1.0.0

1. Now go to **mine** and add **home** and **school** as dependencies like below and run the program.

example:

name: fully

description: A sample command-line application.

version: 1.0.0

environment:

  sdk: '>=2.14.4 <3.0.0'

publish\_to: none

dependencies:

  home:

    path: 'packages/home'

  school:

    path: 'packages/school'

dev\_dependencies:

  lints: ^1.0.0

*notice that* ***description****,* ***version*** *and* ***publish\_to*** *were added.* ***description*** *is a small guide to someone the want to use ur package,* ***version*** *is the version of mine and* ***publish\_to*** *is needed before u can add dependencies else warning will be shown.*

1. You are now free to import home and school to mine as package.

like below.

import 'package:home/output.dart';

import 'package:school/output.dart';

1. **../ :**

This is used to leave the most recent folder to the folder ontop of it.

**../../** is used to leave a folder to the next 2 top folders.

Example:

if **lib/src/form/names.dart**, if I want to import a file in **src** from **name.dart**, I will use **“../file.dart”** (**file.dart** is in **src** folder like **lib/src/file.dart**).

if I want to import a file in **lib** folder from **name.dart**, I will use **“../../full.dart”** (**full** is in **lib** folder like **lib/full.dart**.)

1. But **instead of using** *../ or ../../ or ../../../* etc. **use** “package:src/file.dart” or “package:packageName/full.dart”. it will help you not to confuse your or dart analyzer.
2. .

**21.**

**ISOLATE AND GENERATORS**

**Isolates:**

1. Dart **Isolate** is a component in which dart codes run.
2. Dart is a **single thread programming language**. This **single thread** is called the **mutator thread**.
3. You can create many **isolate** that will help you run your program in parallel. Like when your code have a part that will take a lot of time to complete, you can assign those part to a different isolate that will help you run it while the rest of your code keeps running.
4. Each isolate have it’s own memory, thus it is RAM costly but Dart developers promised to release isolate group by dart **version 2.15.0** whereby many isolate will share the same memory. This will result to many efficiencies.
5. In Dart, we have both **Event queue** and **Microtask queue**.
6. Microtask queue is not really another thread but a mean of seeing that there are some element in the single thread that have higher priority than others.
7. microtask events have higher priority compared to Future events.

example:

void main(List<String> args) {

  print("A");

  Future(() {

    print("B");

    Future.delayed(Duration(seconds: 5), () => print("C"));

    Future.microtask(() => print("D"));

    Future(() => print("E"));

    print("F");

  });

  print("G");

}

The output of the above code are:

**A G B F D E C** but if there was no delay, it will be **A G B F D C E** notice the **D** always comes before **C**, this is because it is a microtask.

1. .
2. Using isolates, your Dart code can perform multiple independent tasks at once, using additional processor cores if they’re available. Isolates are like threads or processes, but each isolate has its own memory and a single thread running an event loop.
3. .

**Generators:**

When you need to lazily produce a sequence of values, consider using a *generator function*. Dart has built-in support for two kinds of generator functions:

* **Synchronous** generator: Returns an [**Iterable**](https://api.dart.dev/stable/dart-core/Iterable-class.html) object.
* **Asynchronous** generator: Returns a [**Stream**](https://api.dart.dev/stable/dart-async/Stream-class.html) object.

To implement a **synchronous** generator function, mark the function body as **sync\***, and use **yield** statements to deliver values:

Iterable<int> naturalsTo(int n) sync\* {

int k = 0;

while (k < n) yield k++;

}

To implement an **asynchronous** generator function, mark the function body as **async\***, and use **yield** statements to deliver values:

Stream<int> asynchronousNaturalsTo(int n) async\* {

int k = 0;

while (k < n) yield k++;

}

If your generator is recursive, you can improve its performance by using yield\*:

Iterable<int> naturalsDownFrom(int n) sync\* {

if (n > 0) {

yield n;

yield\* naturalsDownFrom(n - 1);

}

}

1. **yield** is used to return A in **Type<A>** and **yield\*** is used to return **Type<A>**.

**22.**

**SYNCHRONOUS**

1. A **synchronous operation** blocks other operations from executing until it completes and **synchronous function** only performs synchronous operations.
2. **Iterable(Sychronous Generator):**

* To implement a **synchronous** generator function, mark the function body as **sync\***, and use **yield** statements to return value of **T** in **Iterable<T>**.

Example:

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

* If your generator is recursive or you want to return a type of **Iterable<T>**, you can improve its performance by using **yield\*** instead of **yield**.

Example:

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  yield\* \_showNegativeGenerator(n);

  print("Generator End!");

}

Iterable<int> \_showNegativeGenerator(int n) sync\* {

  print("Negative Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Negative Generator End!");

}

* **.**

1. Go to **Iterable** under **collection chapter above** To learn more.

**23.**

**ASYCHRONOUS**

1. Once initiated, an **asynchronous operation** allows other operations to execute before it completes and an **asynchronous function** performs at least one asynchronous operation and **can also** perform *synchronous* operations.
2. The dart:async library works in both web apps and command-line apps.
3. asynchronous computations usually provide their result as a **Future** or, if the result has **multiple parts**, as a **Stream**.
4. A **synchronous function** can not return the value of an **asynchronous function** unless it is call inside an asynchronous function and it needs to call **await** keyword before the asynchronous function it want to return it’s value.

Example:

Future<String>? fetchUserOrder() {

  return Future.delayed(

    const Duration(seconds: 2),

    () => "Large Latte",

  );

}

void main(List<String> args) async {

  print('Fetching User Order...');

  print("Your Order is:\t${await fetchUserOrder()}");

}

Notice that **await** proceeds the asynchronous function call and that **main** was made **asynchronous** by proceeding it with **async** ( which is now **Future<void> main…**). **await** didn’t proceed print(…) because it is not an asynchronous function. Also, if u remove the await, the asynchronous function will not return completely(the value) but will return only the reference***( Your order is: Instance of ‘Future<String>’)*** to the asynchronous function.

1. What’s the output?

Future<void> printOrderMessage() async {

  print('Awaiting user order...');

  var order = await fetchUserOrder();

  print('Your order is: $order');

}

Future<String> fetchUserOrder() {

  // Imagine that this function is more complex and slow.

  return Future.delayed(const Duration(seconds: 4), () => 'Large Latte');

}

void main() {

  countSeconds(7); //

  await printOrderMessage();

}

// Used to Conut seconds waited...

void countSeconds(int s) {

  for (var i = 1; i <= s; i++) {

    Future.delayed(Duration(seconds: i), () => print(i));

  }

}

After running the code above, try reversing countSeconds(7) and the line below it, then run the code again. What do you found out?.

1. **await:**

* It is used to make every operation below it to wait until it finish it’s own execution.
* Any function that will have **await** must have **async** keyword before the function’s body.

1. **async:**

* It must proceed any function that will implement await and this function must be an asynchronous function or main function.

1. **Future:**
2. A future represents the result of an asynchronous operation, and can have **two states**: **uncompleted** or **completed**.
3. **Uncompleted Future**: When you call an asynchronous function, it returns an uncompleted future, like a reference to the asynchronous function. That future is waiting for the function’s asynchronous operation to finish or to throw an error.
4. **Completed Future**: If the asynchronous operation succeeds, the future completes with a value. Otherwise it completes with an error.
5. A future of type **Future<T>** completes with a value of type **T**. For example, a future with type **Future<String>** produces a **String** value. If a future doesn’t produce a usable value, then the future’s type is **Future<void>**
6. You can wrap **Future** inside a **try catch block** but Future provides a function( **catchError(…)**) used preferably to handle this.

Example:

Future<int> future = getFuture();

future.then((value) => handleValue(value))

.catchError((error) => handleError(error));

**then(…)//** is used to run some codes after the main future code have finished.

1. .
2. **VALUEABLES:**

[**Future**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.html)**(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T> computation())**

Creates a future that contains the result of calling computation

[**Future.delayed**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.delayed.html)**(**[**Duration**](https://api.dart.dev/stable/2.17.3/dart-core/Duration-class.html)**duration, [**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T> computation()?])**

Creates a future that runs its computation after delaying for the specified duration.

[**Future.error**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.error.html)**(**[**Object**](https://api.dart.dev/stable/2.17.3/dart-core/Object-class.html)**error, [**[**StackTrace**](https://api.dart.dev/stable/2.17.3/dart-core/StackTrace-class.html)**? stackTrace])**

Creates a future that completes with an error and an optional StackTrace.

[**Future.microtask**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.microtask.html)**(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T> computation())**

Creates a future containing the result of calling computation asynchronously with [scheduleMicrotask](https://api.dart.dev/stable/2.17.3/dart-async/scheduleMicrotask.html).

[**Future.sync**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.sync.html)**(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T> computation())**

Returns a future containing the result of immediately calling computation.

[**Future.value**](https://api.dart.dev/stable/2.17.3/dart-async/Future/Future.value.html)**([**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T>? value])**

Creates a future completed with value.

[**asStream**](https://api.dart.dev/stable/2.17.3/dart-async/Future/asStream.html)**()** → [Stream](https://api.dart.dev/stable/2.17.3/dart-async/Stream-class.html)<T>

Creates a [Stream](https://api.dart.dev/stable/2.17.3/dart-async/Stream-class.html) containing the result of this future.

[**catchError**](https://api.dart.dev/stable/2.17.3/dart-async/Future/catchError.html)**(**[**Function**](https://api.dart.dev/stable/2.17.3/dart-core/Function-class.html)**onError, {**[**bool**](https://api.dart.dev/stable/2.17.3/dart-core/bool-class.html)**test(**[**Object**](https://api.dart.dev/stable/2.17.3/dart-core/Object-class.html)**error)?})** → [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<T>

Handles errors emitted by this [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html).

[**then**](https://api.dart.dev/stable/2.17.3/dart-async/Future/then.html)**<R>(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<R> onValue(T value), {**[**Function**](https://api.dart.dev/stable/2.17.3/dart-core/Function-class.html)**? onError})**→ [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<R>

Register callbacks to be called when this future completes. The **onError** optional-named-parameter can be used to handle errors but use **catchError (…)** above preferably.

[**timeout**](https://api.dart.dev/stable/2.17.3/dart-async/Future/timeout.html)**(**[**Duration**](https://api.dart.dev/stable/2.17.3/dart-core/Duration-class.html)**timeLimit, {**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<T> onTimeout()?})** → [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<T>

Time-out the future computation after **timeLimit** has passed. **onTimeout()** is used to add what will be printed out if the future delays more than **timeLimit**.

[**whenComplete**](https://api.dart.dev/stable/2.17.3/dart-async/Future/whenComplete.html)**(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<void> action())**→ [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<T>

Registers a function to be called when this future completes. normally used in a case when you want to add more functionality.

[**any**](https://api.dart.dev/stable/2.17.3/dart-async/Future/any.html)**<T>(**[**Iterable**](https://api.dart.dev/stable/2.17.3/dart-core/Iterable-class.html)**<**[**Future**](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)**<T>> futures)** → [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<T>

It is a **static method** that returns the result of the first future in futures to complete.

[**doWhile**](https://api.dart.dev/stable/2.17.3/dart-async/Future/doWhile.html)**(**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**<**[**bool**](https://api.dart.dev/stable/2.17.3/dart-core/bool-class.html)**> action())**→ [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)

It is a **static method** that performs an operation repeatedly until it returns false.

[**forEach**](https://api.dart.dev/stable/2.17.3/dart-async/Future/forEach.html)**<T>(**[**Iterable**](https://api.dart.dev/stable/2.17.3/dart-core/Iterable-class.html)**<T> elements,**[**FutureOr**](https://api.dart.dev/stable/2.17.3/dart-async/FutureOr-class.html)**action(T element))** → [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)

It is a **static method** that performs an action for each element of the iterable, it turn.

[**wait**](https://api.dart.dev/stable/2.17.3/dart-async/Future/wait.html)**<T>(**[**Iterable**](https://api.dart.dev/stable/2.17.3/dart-core/Iterable-class.html)**<**[**Future**](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)**<T>> futures, {**[**bool**](https://api.dart.dev/stable/2.17.3/dart-core/bool-class.html)**eagerError = false, void cleanUp(T successValue)?})** → [Future](https://api.dart.dev/stable/2.17.3/dart-async/Future-class.html)<[List](https://api.dart.dev/stable/2.17.3/dart-core/List-class.html)<T>>

It is a **static method** that waits for multiple futures to complete and collects their results.

1. **Stream(asynchronous Generator):**
2. Streams works like Iterable in that it returns 0,1 or more values, but unlike Iterable, Streams are asynchronous in returning there value.

Example:

void main(List<String> args) {

  Stream.periodic(Duration(seconds: 1), (x) => x).listen((print));

}

This will print **0, 1, …, n** one after the other every **one second** till you terminate the code. This was possible with the **listen(…)** method.

void main(List<String> args) {

  Stream.periodic(Duration(seconds: 1), (x) => x).listen((print));

  Stream.periodic(Duration(seconds: 1), (x) => -x).listen((print));

}

The will print 0,0,1,-1,2,-2, …, n,-n two opposite values at a time each second till you terminate the program.

void main(List<String> args) {

  Stream.fromFutures([Future(() => 3), Future.value(2)]).listen((print));

}

This will print:

2

3

**2** is before **3** because u still remember that **Future.value** executes faster than ordinary **Future**.

1. When you need to get values from a Stream, you have two options:

* Use async and an asynchronous for loop (await for).
* Use the Stream API, as described [in the library tour](https://dart.dev/guides/libraries/library-tour#stream).

1. To get each value as it arrives, either use ***await for*** or subscribe to the stream using the ***listen()*** method.

**Example:**

submitButton.onClick.listen((e) {

// When the button is clicked, it runs this code.

submitData();

});

1. .
2. Stream(Asynchronous generator):

* You consume a stream either using an **await for loop**, which is available inside an **async** or **async\*** function, or by forwarding its events directly using **yield\*** inside an **async\*** function.

Example:

Stream<T> optionalMap<T>(

Stream<T> source , [T Function(T)? convert]) **async**\* {

**if** (convert == **null**) {

**yield**\* source;

} **else** {

**await** **for** (**var** event **in** source) {

**yield** convert(event);

}

}

}

* To implement an **asynchronous** generator function, mark the function body as **async\***, and use **yield** statements to return value of **T** in **Stream<T>**.

Example:

Stream<int> showGenerator(int n) async\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Generator End!");

}

* If your generator is recursive or you want to return a type of **Stream<T>**, you can improve its performance by using **yield\*** instead of **yield**.

Example:

Iterable<int> showGenerator(int n) sync\* {

  print("Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  yield\* \_showNegativeGenerator(n);

  print("Generator End!");

}

Iterable<int> \_showNegativeGenerator(int n) sync\* {

  print("Negative Generator start:");

  for (var i = 1; i < n; i++) {

    print(i);

    yield i;

  }

  print("Negative Generator End!");

}

1. You call **listen** on a **stream** to tell it that you want to receive events, and to register the callbacks which will receive those events.

When you call **listen**, you receive a [**StreamSubscription**](https://api.dart.dev/stable/2.17.3/dart-async/StreamSubscription-class.html) object which is the active object providing the events, and which can be used to **stop** listening again, or to **temporarily** **pause** events from the subscription.

Example:

  final stream =

      Stream.periodic(const Duration(seconds: 1), (i) => i \* i).take(10);

  final subscription = stream.listen(print);

  //Uncomment the below code to see their funtionalities.

  // subscription.pause();

  // print(subscription.isPaused); // true

  // subscription.resume();

  // print(subscription.isPaused); // false

  // subscription.cancel(); //To cancel the subscription.

1. There are **two kinds** of streams:

* **Single-subscription** streams:

Listening twice on a single-subscription stream is not allowed, even after the first subscription has been canceled because a **single-subscription** stream allows only a **single listener** during the whole **lifetime** of the stream.

single-subscription stream doesn't start generating events until it has a listener.

The stream created by an **async\*** function **is a single-subscription** **stream**, but each call to the function creates a new such stream.

If several listeners want to listen to a single-subscription stream, use [**asBroadcastStream**](https://api.dart.dev/stable/2.17.3/dart-async/Stream/asBroadcastStream.html) to create a broadcast stream on top of the non-broadcast stream.

Single-subscription streams are generally used for streaming chunks of larger contiguous data, like file I/O.

* **Broadcast** streams:

These allows any number of listeners, and it fires its events when they are ready, whether there are listeners or not.

1. The below example uses an async function (checkVersion()) without waiting for a result — a practice that can cause problems if the code assumes that the function has finished executing. To avoid this problem, use the [unawaited\_futures linter rule](https://dart.dev/tools/linter-rules#unawaited_futures).

Example:

void main() async {

checkVersion();

print('In main: version is ${await lookUpVersion()}');

}

1. .

**VALUABLES:**

* **.**

**24.**

**LIBRARIES**

1. By default, Every .dart file is a library
2. Only those class defined in the lib folder can be accessed by other packages.
3. By default, any file you create in the lib folder has **library file\_name**; at it’s top but you can add it explicitly.

Example:

library new\_pro;

export "src/calculate.dart";

class A {}

1. Dart don’t have **class private field**s but have **library(file) private field**. This is the reason why private fields in a class can be accessed outside that class provided that you are still in the same file/library.
2. Can one library be formed out of multiple files?

YES: with **part** and **part of** keywords.

1. The main/primary file/library that other libraries are part of will contain the **part ‘other\_lib\_name’;**
2. Any library that is part of the main/primary library will contain **part of ‘main\_lib\_name’;**

Example:

library new\_pro;

export "src/calculate.dart";

// export keyword must be above part

part 'src/new\_pro.dart';

part 'new\_pro1.dart';

class A {}

only this main/primary library can **import**, **export**, and **other derivatives.**

part of 'new\_pro.dart';

//? Notice that there is no

//! library new\_pro1

//? in this file, this is because new\_pro1 is a part of  new\_pro so only one

//?library exist.

part of 'new\_pro.dart';

//? Notice that there is no

//! library new\_pro1

//? in this file, this is because new\_pro1 is a part of  new\_pro so only one

//?library exist.

1. You can access even private fields & methods in one file from the remaining two file because there are of the same library
2. .
3. Can one library contain multiple libraries?

YES: with the **export** keyword.

Example:

library new\_pro; //? This is the primary library.

export 'src/new\_pro.dart';

export 'new\_pro1.dart';

class A {}

Only this primary class is needed to be imported to use other libraries

library new\_pro1;

import 'src/calculate.dart';

class A {}

Notice that this is a stand alone library

Notice also that it can have and import keyword.

It also have the same import as a fellow sub-library. This means that it is totally different from others, although the same primary library exported them.

it can not have top-level field/method with the same name as it fellow sub-library that is why I created classes with different name for the 2 sub-library.

library new\_pro;

import 'calculate.dart';

class B {}

1. The only required argument to **import** is a URI **(uniform resource identifier)**&*URLs* (**uniform resource locators**)) specifying the library. For built-in libraries, the URI has the special **dart**: scheme. For other libraries, you can use a file system path or the **package**: scheme. The **package**: scheme specifies libraries provided by a package manager such as the pub tool.

Example:

import 'package:test/test.dart';

1. If you import two libraries that have conflicting identifiers, then you can specify a prefix for one or both libraries. For example, if library1 and library2 both have an Element class, then you might have code like this.

Example:

import 'package:lib1/lib1.dart';

import 'package:lib2/lib2.dart' as lib2;

// Uses Element from lib1.

Element element1 = Element();

// Uses Element from lib2.

lib2.Element element2 = lib2.Element();

1. **Importing only part of a library:**

If you want to use only part of a library, you can selectively import the library. For

Example:

// Import only foo.

import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.

import 'package:lib2/lib2.dart' hide foo;

1. **Lazily loading a library**
2. ***Deferred loading*** (also called *lazy loading*) allows a web app to load a library on demand, if and when the library is needed. Here are some cases when you might use deferred loading:

* To reduce a web app’s initial startup time.
* To perform A/B testing—trying out alternative implementations of an algorithm, for example.
* To load rarely used functionality, such as optional screens and dialogs.

1. **Only dart2js supports deferred loading.** Flutter, the Dart VM, and dartdevc don’t support deferred loading. For more information, see [**issue #33118**](https://github.com/dart-lang/sdk/issues/33118) and [i**ssue #27776.**](https://github.com/dart-lang/sdk/issues/27776)
2. To lazily load a library, you must first import it using deferred as.

import 'package:greetings/hello.dart' deferred as hello;

1. When you need the library, invoke **loadLibrary()** using the library’s identifier.

Future greet() async {

await hello.loadLibrary();

hello.printGreeting();

}

1. In the preceding code, the await keyword pauses execution until the library is loaded. For more information about async **and** await, see [**asynchrony support**](https://dart.dev/guides/language/language-tour#asynchrony-support).
2. You can invoke loadLibrary() multiple times on a library without problems. The library is loaded only once.
3. Keep in mind the following when you use deferred loading:

* A deferred library’s constants aren’t constants in the importing file. Remember, these constants don’t exist until the deferred library is loaded.
* You can’t use types from a deferred library in the importing file. Instead, consider moving interface types to a library imported by both the deferred library and the importing file.
* Dart implicitly inserts loadLibrary() into the namespace that you define using deferred as *namespace*. The loadLibrary() function returns a [**Future**](https://dart.dev/guides/libraries/library-tour#future).

1. **DART LIBRARIES:**

Dart has [**a rich set of core libraries**](https://dart.dev/guides/libraries), providing essentials for many everyday programming tasks:

* Built-in types, collections, and other core functionality for every Dart program (**dart:core**)
* Richer collection types such as queues, linked lists, hashmaps, and binary trees (**dart:collection**)
* Encoders and decoders for converting between different data representations, including JSON and UTF-8 (**dart:convert**)
* Mathematical constants and functions, and random number generation (**dart:math**)
* File, socket, HTTP, and other I/O support for non-web applications (dart:io)
* Support for asynchronous programming, with classes such as Future and Stream (**dart:async**)
* Lists that efficiently handle fixed-sized data (for example, unsigned 8-byte integers) and SIMD numeric types (**dart:typed\_data**)
* Foreign function interfaces for interoperability with other code that presents a C-style interface (**dart:ffi**)
* Concurrent programming using isolates — independent workers that are similar to threads but don’t share memory, communicating only through messages (**dart:isolate**)
* HTML elements and other resources for web-based applications that need to interact with the browser and the Document Object Model (DOM) (**dart:html**)

1. .

**25.**

**URIs:**

* + 1. The [**Uri class**](https://api.dart.dev/stable/dart-core/Uri-class.html) provides functions to encode and decode strings for use in URIs also know as URL
    2. To **encode** and **decode** characters except those with special meaning in a URI (such as /, :, &, #), use the ***encodeFull()*** and ***decodeFull()*** methods.

Example:

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

var encoded = Uri.encodeFull(uri);

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

var decoded = Uri.decodeFull(encoded);

assert(uri == decoded);

Notice how only the space between **some** and **message** was encoded.

* + 1. To **encode** and **decode all** of a string’s characters that have special meaning in a URI, including (but not limited to) **/,** **&,** and **:**, use the ***encodeComponent***() and ***decodeComponent***() methods.

Example:

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

var encoded = Uri.encodeComponent(uri);

assert(

encoded == 'https%3A%2F%2Fexample.org%2Fapi%3Ffoo%3Dsome%20message');

var decoded = Uri.decodeComponent(encoded);

assert(uri == decoded);

* + 1. To create a Uri from a string, use the ***parse()*** static method:

Example:

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

assert(uri.scheme == 'https');

assert(uri.host == 'example.org');

assert(uri.path == '/foo/bar');

assert(uri.fragment == 'frag');

assert(uri.origin == 'https://example.org:8080');

* + 1. You can build up a URI from individual parts using the ***Uri() constructor***:

Example:

var uri = Uri(

scheme: 'https',

host: 'example.org',

path: '/foo/bar',

fragment: 'frag');

assert(uri.toString() == 'https://example.org/foo/bar#frag');

* + 1. .

**26.**

**DATE\_TIME:**

1. A DateTime object is a point in time. The time zone is either UTC or the local time zone.
2. You can create DateTime objects using several constructors:

// Get the current date and time.

var now = DateTime.now();

// Create a new DateTime with the local time zone.

var y2k = DateTime(2000); // January 1, 2000

// Specify the month and day.

y2k = DateTime(2000, 1, 2); // January 2, 2000

// Specify the date as a UTC time.

y2k = DateTime.utc(2000); // 1/1/2000, UTC

var y2k = DateTime.utc(2000);

// Add one year.

var y2001 = y2k.add(const Duration(days: 366));

assert(y2001.year == 2001);

// Subtract 30 days.

var december2000 = y2001.subtract(const Duration(days: 30));

assert(december2000.year == 2000);

assert(december2000.month == 12);

// Calculate the difference between two dates.

// Returns a Duration object.

var duration = y2001.difference(y2k);

assert(duration.inDays == 366); // y2k was a leapyear.

***Note:****hours are specified between 0 and 23, as in a 24-hour clock.*

1. The DateTime class provides a constant for each day and month name - for example, [**august**](https://api.dart.dev/stable/2.17.3/dart-core/DateTime/august-constant.html) is 8 and [**friday**](https://api.dart.dev/stable/2.17.3/dart-core/DateTime/friday-constant.html) is 5.
2. Use the [**add**](https://api.dart.dev/stable/2.17.3/dart-core/DateTime/add.html) and [**subtract**](https://api.dart.dev/stable/2.17.3/dart-core/DateTime/subtract.html) methods with a [**Duration**](https://api.dart.dev/stable/2.17.3/dart-core/Duration-class.html) object to create a DateTime object based on another.

Example:

  final now = DateTime.now();

  final later = now.add(const Duration(hours: 36));

  print("$now and 36 hours later($later).");

1. To find out how much time is between two DateTime objects use [**difference**](https://api.dart.dev/stable/2.17.3/dart-core/DateTime/difference.html), which returns a [**Duration**](https://api.dart.dev/stable/2.17.3/dart-core/Duration-class.html) object:
2. **DURATION**:

* It represents the difference from one point in time.
* It has only one constructor; **Duration({int day, int hour, …,microseconds})**. This constructors argument can be **negative** or **positive**.
* It has many properties to retrieve it’s value like **inDay**, **inHour**…etc
* It has many operators to manipulate it like, \*, +, ~/, … etc.

1. **STOPWATCH**:

* It have only one constructor; Stopwatch();
* It has many properties like elapsed, elapsedTicks, elapsedMillisecond, …etc
* it has many methods like start(), stop(), reset().
* start();// starts the stopwatch
* stop();// pauses the stopwatch
* reset();//re-starts the stopwatch
* It also have an operator; ==

1. .

**27.**

**ACCESS MODIFIERS:**

1. **Private Variables:**
2. **Final Variables:**
3. A final variable can be set only once.
4. Instance variable **can’t** be **const** but **final** or either **static const** but not just **const**.
5. final variables are only stored from their first use, thus they are not compile time constant.
6. **Final instance variables** must be initialized before the constructor body starts — at the **variable declaration**, by a **constructor parameter**, or in the **constructor’s**[**initializer list**](https://dart.dev/guides/language/language-tour#initializer-list).
7. If you need to assign the value of a **final instance variable** after the constructor body starts, you can use one of the following:

* Use a [**factory constructor**](https://dart.dev/guides/language/language-tour#factory-constructors).
* Use **late final**, but [be careful:](https://dart.dev/guides/language/effective-dart/design#avoid-public-late-final-fields-without-initializers) a late final without an initializer adds a setter to the API.

1. Although a **final** object **cannot be modified**, its **fields can be changed**. In comparison, a **const object and its fields cannot be changed**: they’re immutable.
2. .
3. **Const Variables:**
4. **const variables** are implicitly final.
5. Instance variable **can’t** be **const** but **final** or either **static const** but not just **const**.
6. **const** variable are **compile-time constant**—It is initialize at compile time(immediately after declaration)-- , so if the variable is at class-level, mark it with **static const** instead.
7. **const** is not only meant to proceed types but can proceed value and can also be used to create constructor that needs constant values.

Example:

const x2 = x\*2;

var a = const A(); // here a can be changed but can’t be changed

const list = []; // here both list and [] can’t be changed.

var list1 = const[]; // here list1 can be changed but [] can’t be changed.

1. You can define constants that use[**type checks and casts**](https://dart.dev/guides/language/language-tour#type-test-operators)**(is**and**as)**, [**collection if**](https://dart.dev/guides/language/language-tour#collection-operators), and [**spread operators**](https://dart.dev/guides/language/language-tour#spread-operator)**(... and ...?)**,

Example:

  const Object i = 3; // Where i is a const Object with an int value...

  const list = [i as int]; // Use a typecast.

  const map = {if (i is int) i: "int"}; // Use is and collection if.

  const set = {if (list is List<int>) ...list}; // ...and a spread.

1. **.**
2. **Static variables:**

Static methods (class methods) don’t operate on an instance, and thus **don’t** have access to **this**. They do, however, have access to static variables.

1. **Instance variables:**
2. Instance variables can be **final** but **not const**, unless it is **static const**.
3. **instance variable** when **not nullable** nor **late**, most not be assigned in the body of any constructor in the class.

Example:

class A {

  int e;

  A() : e = 5; //OR A(this.e); Not A(){e = 5}

}

1. **All instance variables** generate an **implicit getter** method. Non**-final instance variables** and **late final instance variables** without initializers also generate an **implicit setter** method.
2. **Top-level Variable:**
3. **Local variable:**

**28.**

**GARBGE COLLECTOR:**

**29.**

**VARIABLES:**

1. **int (int I = 0xDEADBEEF;):**
2. Integer values number are not larger than 64 bits, depending on the platform. On the Dart VM, values can be from -263 to 263 - 1. Dart that’s compiled to JavaScript uses [**JavaScript numbers,**](https://stackoverflow.com/questions/2802957/number-of-bits-in-javascript-numbers/2803010#2803010) allowing values from -253 to 253 - 1.
3. **Valuables**:
4. **int exp = 8e3; or 8e+3;**// This is how to declare Exponentials.
5. **int i = int.parse(‘1’)**;//used to convert from string to int. To add the **base** of the **int**, use **radix parameter** like **int.parse(‘42’, radix: 16)** to show that 42 is in base 16.
6. **exp.sign;** // This is used to get the sign of the integer stored in “exp”.
7. **exp.isOdd;**
8. **exp.isEven;**
9. **exp.ads();** // it is used to get the absolute values of “exp”.
10. **1.2345.toStringAsFixed(2)**// will produced 1.23; that’s to 2 decimal places.
11. **123.456.toStringAsPrecision(2)**// will output 120; that’s to 2 significant figure.
12. **double( double d = 1.5e2;):**
13. 64-bit (double-precision) floating-point numbers, as specified by the IEEE 754 standard
14. **Valuables**:
15. double db = double.parse(‘1’);//used to convert from string to double.
16. db.round();
17. db.truncate();
18. db.toString();
19. db.runtimeType();
20. String pi = 3.1415.toStringAsFixed(2);// makes pi to be a string //that have only “3.14”.
21. db = 3.142342.toStringAsFixed(2); // converts the double value to 2 fixed point.
22. **String:**
23. A Dart string is a sequence of UTF-16 code units. You can use either single or double quotes to create a string.
24. When **r** proceed a string literal, the string will be printed raw.

Example:

main() {

  print(r'Love first sign is:\tI know?');

}

This will print:

*Love first sign is:\tI know*

1. Multi-line String is put inside three quotation marks; that’s either **single quote** or **double quote**. This String will maintain it’s space.
2. Strings are interpolated with **dollar sign** to change other types to string;

Example:

int n = 4;

String s = “$n Love”; // it is equivalent to “4 Love”.

1. To get the string corresponding to an object, Dart calls the object’s **toString()** method.
2. You can **concatenate** strings using **adjacent string literals** or the **+ operator**.

Example:

  var s1 = 'String '

    'concatenation'

    " works even over line breaks.";

1. Strings can be accessed like **list**.

Example:

void main(List<String> args) {

  String str = 'Nnamdi';

  print(

    str[3],

  );

Or

print(

   'Nnamdi'[3],

  );

}

1. Strings are immutable objects thus in String API, none of the methods actually changes the state of a String. For example, the method replaceAll() returns a new String without changing the original String.

Example:

  var greetingTemplate = 'Hello, NAME!';

  var greeting = greetingTemplate.replaceAll(RegExp('NAM'), 'Bob');

// greetingTemplate didn't change.

  assert(greeting != greetingTemplate);

1. A **StringBuffer** **doesn’t generate** a new String object until **toString()** is called. The **writeAll()** method has an optional second parameter that lets you specify a separator. Example:

  var sb = StringBuffer();

  sb

    ..write('Use a StringBuffer for ')

    ..writeAll(['efficient', 'string', 'creation'], ' ')

    ..write('.');

  var fullString = sb.toString();

  assert(fullString == 'Use a StringBuffer for efficient string creation.');

1. .

**Valuables**:

1. String str = 1.toString();
2. String n = “Nnamdi EarnDee”.toUpperCase();
3. **str.codeUnits**// Get all the UTF-16 code units in the string.

var codeUnitList = 'Never odd or even'.codeUnits.toList();

assert(codeUnitList[0] == 78);

1. n.allMatches(String,[startIndex]);
2. str.codeUnitAt(1);
3. str.compareTo(n);
4. n.contains(“Earn”);
5. n.endsWith(“Dee”);
6. str.hashcode;
7. n.indexOf(“Dee”);//
8. isEmpty;
9. isNotEmpty;
10. str.length;
11. matchAsPefix(…);
12. padLeft(...);
13. padRight(...);
14. replaceAll(...);
15. split(…);// Split a string using a string pattern.

var parts = 'progressive web apps'.split(' ');

assert(parts.length == 3);

assert(parts[0] == 'progressive');

Or

for (final char in 'hello'.split('')) {

print(char);

}

1. n.substring(starting index, ending index);
2. n.startWith(‘Nnamd’);
3. str.toUppercase;
4. **Trim()//**assert(' hello '.trim() == 'hello');
5. Trimleft();
6. TrimRight();
7. **bool:**
8. **Valuables**:
9. **num:**
10. Both **int** and **double** are subtypes of [**num**.](https://api.dart.dev/stable/dart-core/num-class.html)
11. The num type includes basic operators such as +, -, /, and \*, and is also where you’ll find **abs()**, **ceil()**, and **floor()**, among other methods.
12. Bitwise operators, such as >>, are defined in the **int** class.
13. If num and its subtypes don’t have what you’re looking for, the [**dart:math**](https://api.dart.dev/stable/dart-math) library might.
14. **Valuables**:
15. **Null:**
16. **Never:**

Indicates that an expression can never successfully finish evaluating. Most often used for functions that always throw an exception.

1. **Runes and grapheme:**

In Dart, [runes](https://api.dart.dev/stable/dart-core/Runes-class.html) expose the Unicode code points of a string.

**const** string = 'Dart';

**final** runes = string.runes.toList();

print(runes); *// [68, 97, 114, 116]*

Runes and grapheme clusters

In Dart**,**[**runes**](https://api.dart.dev/stable/dart-core/Runes-class.html) expose the Unicode code points of a string. You can use the [**characters package**](https://pub.dev/packages/characters)to view or manipulate user-perceived characters, also known as [**Unicode (extended) grapheme clusters**.](https://unicode.org/reports/tr29/#Grapheme_Cluster_Boundaries)

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world’s writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing Unicode code points within a string requires special syntax. The usual way to express a Unicode code point is \uXXXX, where XXXX is a 4-digit hexadecimal value. For example, the heart character (♥) is \u2665. To specify more or less than 4 hex digits, place the value in curly brackets. For example, the laughing emoji (😆) is \u{1f606}.

If you need to read or write individual Unicode characters, use the characters getter defined on String by the characters package. The returned [**Characters**](https://pub.dev/documentation/characters/latest/characters/Characters-class.html) object is the string as a sequence of grapheme clusters. Here’s an example of using the characters API:

1. **Symbols:**
2. if **(DateTime.now().hour < 12)**// used to get if current hour is before 12 noon.
3. **.**
4. **Unicode special characters:**

Unicode defines a unique numeric value for each letter, digit, and symbol used in all of the world’s writing systems. Because a Dart string is a sequence of UTF-16 code units, expressing Unicode code points within a string requires special syntax. The usual way to express a Unicode code point is \uXXXX, where XXXX is a 4-digit hexadecimal value.

Example:

1. Heart/love character ---> \u2665
2. laughing emoji ---> \u{1f606}
3. @ ---> \u{ff20}
4. ! --->\uff01 etc
5. treble clef --> (‘\u{1D11E}’)

**VALUABLES:**

1. **HASHCODE**:

* To generate a single hash code for multiple properties of an object, you can use **Object.hash()**. To generate a hash code for a collection, you can use either **Object.hashAll()**( if element order matters) or **Object.hashAllUnordered()**.

Example: …

* Try and override **== operator** if hashcode is overridden

Example:

class Person {

final String firstName, lastName;

Person(this.firstName, this.lastName);

// Override hashCode using the static hashing methods

// provided by the `Object` class.

@override

int get hashCode => Object.hash(firstName, lastName);

// You should generally implement operator `==` if you

// override `hashCode`.

@override

bool operator ==(dynamic other) {

return other is Person &&

other.firstName == firstName &&

other.lastName == lastName;

}

}

void main() {

var p1 = Person('Bob', 'Smith');

var p2 = Person('Bob', 'Smith');

var p3 = 'not a person';

assert(p1.hashCode == p2.hashCode);

assert(p1 == p2);

assert(p1 != p3);

}

* DONE!

1. **STRINGBUFFER:**

* Allows for the incremental building of a string using ***write\*()*** methods.
* The strings **are concatenated** to a **single string only** when [**toString**](https://api.dart.dev/stable/2.17.1/dart-core/StringBuffer/toString.html)**()** is called.
* To add a newline after the object's string representation, use [**writeln**](https://api.dart.dev/stable/2.17.1/dart-core/StringBuffer/writeln.html)**()**.
* To write multiple objects to the buffer, use [**writeAll**](https://api.dart.dev/stable/2.17.1/dart-core/StringBuffer/writeAll.html)**()**.
* To clear the buffer, so that it can be reused, use [**clear**](https://api.dart.dev/stable/2.17.1/dart-core/StringBuffer/clear.html)**()**.

Example:

  final buffer = StringBuffer('DART'),

name = StringBuffer("Nnamdi");

  print(buffer.length); // 4

  buffer.write(' is open source since 2011');

  print(buffer); // DART is open since 2011

  buffer.writeln(); // Contains "DART is open source

//since 2011\n".

  buffer.writeln('+' \* (buffer.length - 1));

//Writes the String

  //the number of times you specified.

  print(buffer.length); // 62

  const separator = '#';

  buffer.writeAll(['Dart', 'is', 'fun!'], separator);

  print(buffer.length); // 74

  print(buffer);

  final text = buffer.toString();

  print(text);

  buffer.clear();

  print(buffer.isEmpty); // true

  print(buffer.length);

1. **REGEXP:**

Regular expressions are patterns/model, which can be used to mach string or part of the string. Example:

**The fat cat ran down the street. It was searching for a mouse to eat.**

* 1. **g** means global and it is used to search all similar character or word in a sentence. like:
* /cat/**g** is used to search for all the place that
  1. .

Regular expressions is used for efficient searching and pattern matching of strings.

* **allMatches(String,[int start = 0]);//** Matches this pattern against the string repeatedly.
* **escape(String text);//** Creates a RegExp syntax that matches text.
* **firstMatch(String);//** Finds the first match of the regular expression in the string input.
* **hasMatch(String);//**
* **isCaseSensitive;**
* **isDotAll;** Whether "." in this regular expression matches line terminators.
* **isMultiline;**
* **isUnicode;**
* **matchAsPrefix(String, [int start = 0]);//** Matches this pattern against the start of string.
* **pattern;** The source regular expression string used to create this RegExp.
* **stringMatch(String);//** The substring of the first match of this regular expression in input.
* **.**

1. **MATCHER:**

* A result from searching within a string.
* A Match or an [**Iterable**](https://api.dart.dev/stable/2.17.1/dart-core/Iterable-class.html) of Match objects is returned from the [**Pattern**](https://api.dart.dev/stable/2.17.1/dart-core/Pattern-class.html) matching methods ([**Pattern.allMatches**](https://api.dart.dev/stable/2.17.1/dart-core/Pattern/allMatches.html) and[**Pattern.matchAsPrefix**](https://api.dart.dev/stable/2.17.1/dart-core/Pattern/matchAsPrefix.html)).
* DONE!

1. **PATTERN:**

* **allMatches(String,[int start = 0]);//**

Matches this pattern against the string repeatedly. This begins from the start index you specified, which is **“0”** by default.

* **matchAsPrefix(String, [int start = 0]);//**

Matches this pattern against the start of string. This begins from the start index you specified, which is **“0”** by default.

**Example:**

1. **COMPARING:**

* Implement the [**Comparable**](https://api.dart.dev/stable/dart-core/Comparable-class.html) interface to indicate that an object can be compared to another object, usually for sorting. The compareTo() method **returns < 0 for smaller**, **0 for the same**, and **> 0 for bigger**.

1. **.**

**ADVICE:**

1. **Be consistent.**

**Your code should maintain the same pattern.**

1. **Be brief:**

Your code should be concise( economical not dense).

1. You can use a block comment (/\* ... \*/) to temporarily comment out a section of code, but all other comments should use //.

**OTHERS:**

V8 is Google’s open source high-performance JavaScript and WebAssembly engine, written in C++. It is used in Chrome and in Node.js, among others. It implements [ECMAScript](https://tc39.es/ecma262/) and [WebAssembly](https://webassembly.github.io/spec/core/), and runs on Windows 7 or later, macOS 10.12+, and Linux systems that use x64, IA-32, ARM, or MIPS processors. V8 can run standalone, or can be embedded into any C++ application.

ECMAScript is **a Standard for scripting languages such as JavaScript, JScript, etc**. It is a trademark scripting language specification. JavaScript is a language based on ECMAScript.

WebAssembly is an open standard that allows the execution of binary code on the web. This standard, or format code, lets developers bring the performance of languages like C, C++, and Rust to the web development area. This technology is commonly used **to perform demanding operations in the browser**.