- DART

  - created by google (oct 10, 2011)
  - open-source
  - statically typed language.
    - compile code (in JS comparison)
    - every variable has a type

* flutter appeared in may 2017 - by google.

its a UI SDK.

Software Development Kit

  - type inference
  - multi-platform
  - easy to learn
  - concise - less ceremony.
    - less beaurocracy (in JAVA compare)
  - modern ideas
    - (promises) - named future
  - null safety
    - ability to express, if we accept or not.
    - Java, C#, PHP, you create variable & not set value to it it stores null.

it is semantics of your code, reduce bugs.
Dart, Swift, Kotlin, Go supports null-safety.

- original intention to replace JS
- current intention - front-end apps
  - angular Dart
  - flutter

- inspired by C, Java, JavaScript, Erlang, SmallTalk, Swift/Kotlin (Dart 2)
- compiles to
  - native code
  - intermediate language
  - JavaScript

- every Dart app has main function (similar to C, Java)
- full OOP language with type interface. (everything will be an object)

- has features from many languages:
  - extensions
  - mixins
  - futures (async programming, similar to promises.)

- when compiling to web:
  the code is going to be executed in JavaScript ~~Utilization~~ Virtual machine

- LANGUAGE TYPES —

developer write code		ship in	eg.
interpreted lang	→	Source Code	TS
intermediate lang	→	Byte Code	Java
compiled lang	→	Machine Code	Dart

Dart language

- compiles to → JS Source code
  → Byte Code
  → Machine Code

- in web Apps
  JS Source Code is used.

- in Desktop OS's
  windows, Linux, MacOS
  Bytecode is used

- in Android / iOS
  machine code is used.

* in iOS, it is forbidden to have Virtual Machine.
(commercial issue from Apple)

- var x = 1; (here int x)
  - → inferring the type of variable, based on value.
  - → can't change it later on like JS.

- print(x); – sends a message to console with \n after.
  - – on Android/iOS – output console, not screen
  - – on web – the browser's console.
  - – on Desktop – the output console.

- no global code – everything must be in
  - – class;
  - – function.

- no function keyword.
  - – eg. name() { ---
    }

- everything is public in package, unless its prefixed –
  everything that we are doing in project
  is by default a package.
  → kind of a folder

when you want to use something from other package, you need to import the package.

(similar to packages in Java
— namespaces in .NET, PHP)

- So PACKAGE ?

its a way to group, to contain different objects, classes, functions or global variables

- for (var i = 0; i < 10; i++) { }

for ((var i in [1,2,3])) { print(i); }

similar to for of - loop of other lang.
similar to for each

— in JS for-in will not loop through collections it loops through properties of your object.

- type inference — if you don't want to define a type everytime, you can use var, but it should be assigned something.

eg.-

```
int data = 3;
var otherData = 5;
```

] mutable variables

```
var x;
```

→ variable not assigned something, it becomes type dynamic.

- **final**

  - keyword to make variables

  eg. -

```
final notMutableObj = "final String";
final int otherInt = 5;
```

] immutable variables

  - cannot be changed after its set, on the first time

- Difference b/w const & immutable variable -

- constant - we define & VALUE we set at compilation time)

- immutable variable - variable, place in memory while our app is running, after it was set, it cannot be changed.
  - you don't need value at compile time.

- core Data Types in Dart –

everything is an OBJECT

String	String Values ;
int	integar Values ;
double	double Values ;
bool	boolean Values ; (yes / no)
dynamic	anything ;
num	a Number, you don't define type of.

– num (generic version of a number)

int double

- creating multiline strings –

```
var multilineString = " " "
hi, there,
how are you,
okay.
" " " ;
```

similar to Kotlin and Swift languages. ` ` in JS.

- int is full object, there's a class behind

↓ so

it has methods & properties

* want to see inside <u>ctrl + Click</u> on it.

it is class.

↓ so

it has methods, constructors, & everything of OOP.

- <u>Collections</u> –

Similar to JSON Syntax.

– List<Type> name = ["—", "—"];

eg. –

List<String> countries = ["Argentina", "Mexico"];

kind of arrays.

- sets — use curly braces for the list.
  - does not accept repetitions/duplicates.
  - no actual order inside.

eg. —

```
var strings = {"a","b","c"};
```

- maps (similar to dict. or object in JSON)
  - key : value
  - difference is on type.

eg. —

```
var readWriteMap = {"foo": 1, "bar": 2};
```

↓ shows Map<String, int>

- in Dart the key doesn't need to be strings

eg. —

```
var json = {
  "name": "Jane",
  "age": 34
};
```

$\xrightarrow{\text{show}}$ Map<String, object>

* Dart and Flutter default guidelines for multiline is trailing commas (,).

* Object is similar dynamic,
but dynamic accepts also something that might not be an object (exceptions null)
↓
null value is not an object.

* null safety - is not default in Dart.
- is default in Flutter.

eg. -

String myName;
↳ no compilation happens.

* value of type null can't be assigned to a variable of type String.

eg. - late String myName;
↳ prefix, i don't have a value now, but i will assign it later.

- define semantic null level type

eg. —

String ? myLastName ;

↓

it's a null-level String, so it can be string or it can be null.

— explicitly say it in type declaration

(similar to Java or C# default ~~option~~ object, it can accept null)

eg. — print (myLastName.length);

→ shows compilation error.

→ You can't get length of null-level String.

↓ WHY?

because it might be null.

* if its a null

we can't access length

method or

property

we get

null pointer exception/
null reference exception

- So, in Dart –

null manual over-ride

final lengthForSure = myLastName!.length

assuming its not null.

to compiler

variable might be null, but it ~~w~~ont be null.
~~Go Ahead & get length~~

if its null, then
run-time error

- Safe call operator ?.

eg. –

var length = myLastName ?. length.

if its null, its not going to execute length, it will ignore whats after the dot.

then, type of length is int?

How do?

if there is a last name, there is an integar, but if no last name, then no integar

So, an optional integar.

* if you are working with UI, flutter framework, we don't use optionals dot.

- **functions**

implicit return type.

- you can specify the type you want to return

void — i'm not returning anything

eg —

```
String aFunctionString(){
    ⋮
    return "I'm a Function";
}
```

- **define named arguements**

* flutter HEAVILY using this.

eg —

```
printCusto (String name, double balance{bool isActive})
{
    print (name, "$balance", isActive);
}

allPoint (){
    printCusto ("name", 200, false);
}
```

optional arguement.

so it becomes a named arguement

OR.

MANDATORY.

{required bool is--}

```
allPrint (){
    printCusto ("Jane", balance:200, isActive:false);
}
```

this is making function call more semantic.

- Lambda or Anonymous Functions -

```
Function myFunction2 (int number, bool isReady) { };
```

or var.

This is a Variable containing a function

you can pass this as an argument,

can recieve 2nd return

parantheses

argument declaration

code block.

(sort of similar to OS anonymous functions)

- **Arrow functions**

eg:- type

Function myArrow1 = (int a, int b) => a+b;

only for one expression

(100% equivalent)
(does same thing)

{ } codeblock for several expressions in LAMBDA functions.

Function myArrow2 = (int a, int b) {
    return a+b;
};

* var age = () => 3;

type

final myArrow3 = (int a, int b) {
    return a+b;
}

* variable → is of type function
mutable variable →
constant →

* VS code shortcut (core+.) for ;.

* LAMBDA expression - in some languages for only functional programming.
(only one)

* void is a data type.
↳ but you can't have a variable of type void.
↳ so can't have have value of type void
↳ so you can't have void objects
↳ void by itself is a type.
↳ types are not normal objects

eg = int is not an object.
↳ you can have objects of type int.

- **Class in Dart.**

eg.

```
class Person {
    int id = 0;          — property
    show() {             — a function — now METHOD.
        print ("Person Id: ${this.id}");
    }
}
```

flutter-way

— **creating Objects**

```
final p = Person();
final q = new Person();
```

same.

is Optional (similar to KOTLIN & SWIFT)

Objects = new instances of a Class.

- Constructor -

eg. -

class name

```
class Request {
    int id = 0;
    String url = " ";
    Request (int id, String url){
        this.id = id;
        this.url = url;
    }
}
```

function with a name of class.

(similar to Java, in JS we use constructor keyword)

- PRIMARY CONSTRUCTOR can receive
  - no arguements.
  - one arguements.
  - more arguements.
  - named arguements.

- can't create another one with same name.

- do only one.

- NAMED Constructors –

eg.

Request.fromData (this.id, this.url);

name of class — dot — name of constructor

* You should start with from

* Request (int id, String url) {

this.id = id;

this.url = url;

}

(i) recieving constructor arguements

(ii) assign these arguements into properties

in OOP,

Shortcut in AAT.

Request (int this.id, String this.url) {

}

Shortcut.

→ Request (this.id, this.url);

Here type is infer from Property Declaration

- an idea of a factory
  - → it can return null.
  - → object of your type
  - → or object of a sub-type

call via static METHOD.

— inheritance (in DART)

eg.

```
class HttpRequest extends Request {
  HttpRequest(int id, String url): super(id, url);
}
```

(similar to Java, JavaScript).

constructor receiving 2 arguements extending from super constructor

how to call?

* if you end with ; you don't have a code block

named parameters in Constructor eg.—

```
class Car {
    int id;
    String name;
    Car ({required this.id, required this.name});
}
```

* Flutter is full of named Constructors everywhere
Same as function

* widget is a UI class → has builder

* Abstract Class → extend from it & create actual concrete class to use
→ you can't create instances of an Abstract Class.

eg.—

```
abstract class AbstractRequest {
}
```