

# \* Flutter \*

## 1<sup>ère</sup> partie - DART

- 2-3 Installation, explication générale,  
débuter un projet
- 4 JavaScript vs Dart
- 5-8 Types de variables/fonctions  
(var, String, int, double, bool, const,  
final, List, Map, void, enum)
- 7 Paramètres (optionnel/required, nommées)
- 7-11 Classes (constructeur, interfaces, abstract,  
héritage → extends/super, get/set)
- 12-14 Import et Export (hide/show, async/await,  
deferred, delayed, type Future) /
- 9 Type Générique <T>

# Flutter & Dart

- ① Installation (suivre vidéo dyma) ...
  - ② Créeer new project et explications

# flutter create my\_first\_app

 Important

\* `pubspec.yaml` = `package.json`  
`(dart)`                    `(js)`

N.B.: format yalm => attention à l'indentation

\* `pubspec.lock` = `package.lock.json`

\* android  
ios  
linux  
macos  
web  
windows } dossiers qu'on n'aura pas vraiment à toucher puisque les infos s'ajouteront d'eux-mêmes au fur et à mesure de l'avancement de l'app

\* **lib** → c'est là-dedans qu'on travaille (on y met les components)  
↓  
**main.dart** = entrée de notre app.

3. Débuter le projet
  - » Lancer **Android Studio** *Android*
  - ou ds Terminal: **open -a Simulator** *Mac*
  - » ds Terminal: **flutter run**
4. Pour mettre à jour le visuel après un changement
  - » ds Terminal: **r** (ça ne change que le visuel et conserve l'état)

N.B: Pour remettre à jour l'état aussi: **R**
5. **main.dart** → doit contenir **main(){}{}**
6. **Components** (ou Widget)
  - ↳ doivent avoir chacun une fonction **build**

# JAVASCRIPT vs DART

console.log()

if (nomVariable)

"opérateurs" → idem

var x = y == 5 ? true : false; → idem

"switchCases" → idem

"boucles" → "for", "for in", "forEach",  
"do", "do while"  
↓

au besoin, voir:  
**break** et **continue**

ClassN x = new ClassN()

ClassN x = ClassN()  
(new = optionnel)

private class cl

static fn() {}

import './Car.js'  
optionnel

( html, css et js  
layout      logique )

class \_cl

static String fn() {}

import './Car.dart'  
obligé!!!

( dart  
layout ET logique )

## 8. Types

Comme en TypeScript,  
Dart est typé

**var** → en Dart ≠ global

↳ Une fois un type assigné, ne peut plus changer de type

mais

**String myString = 'bla';**

Inclure une variable dans une string:

'Se dis **\$myString** !!!'

... ou encore...

**\${myString.toUpperCase()}**

**int myInt = 123;**

**bool myBool = true;**

**double myDouble = 12.1538;**

Constantes

**const int myInt2 = 3;**

**final** ⇒ peut être assigné au moment de rouler une fonction avec des variables

ex: **fn(n1, n2){**

**final int sum = n1 + n2;**

→ équivaut à "promise" en JS

## \* Array

List myList = [1, 2, "Bonjour"]

List<int> myList2 = [4, 5, 6];

(myList.add(8);  
myList.removeAt(0);  
... bref, comme en JS)

## \* Map (objet)

Map myMap = {  
'key1': [1, 2, 3],  
'key2': 'value2'  
}

.length  
.keys .values  
.entries

↳ ajout: myMap['key3'] = 8;

→ type: Map<int> [...]

→ const: const Map [...]

→ spread operator: Map myMap2 = {  
... myMap,  
'key4': 44, 23  
}

N.B: Pour savoir si existe :

myMap.containsKey('key3')

myMap.containsValue('value2')

## 9. Fonctions et typage

**String fnName(String f, [String? last]) {**

... ou encore ...  
[String last = 'default']

N.B: main(){} = void main(){}

10. "Paramètres nommés": par défaut,  
ces paramètres deviennent optionnels

↳ Pour qu'ils soient obligatoires:

**String fnName({required String f, String?**

**last = 'default' }) {**

return f + last;

}

**fnName(f: 'Cinthia', last: 'Letourneau')**

11. Classes et enum (voir capture d'écran ...)

**String get fancyname {**

return 'Mr. \${this.name};'

}

**set name(String newName) {**

this.name = newName;

}

```
enum ModelType {
    modelS,
    modelX
}

class Tesla {
    ModelType model;
    bool autoDrive = false;

    Tesla({ this.model }){}
    Tesla.withAutodrive({ this.model, this.autoDrive = true }){}
}

main() {

    Tesla car = Tesla(model: ModelType.modelS);
    Tesla betterCar = Tesla.withAutodrive(model: ModelType.modelX);
}
```

instances

constructor (raccourci)

constructor "nommé"

## 12. Héritage de classe

```
class Car {  
    String engine;  
    sayHello() => print('yo !');  
    Car({this.engine});  
}
```

Fais référence  
au constructeur  
de la classe  
parent

```
class Tesla extends Car {  
    ModelType model;  
    String name;  
    sayHelloBye() {  
        Super.sayHello();  
        print('Bye');  
    }  
}
```

```
Tesla({this.model, this.name}):super()  
engine:'electric'{ }
```

N.B: Dans le cas où 2 fonctions qui ont le même nom ds parent et ds enfant, pour dire lequel on priorise  
**@override**

## 13. Les Génériques => <T>

**PAS TROP COMPRIS  
à revoir...**

## Les interfaces implicites

En Dart, il n'y a pas de déclaration d'interface, en effet chaque classe définit implicitement une interface.

Cette interface contient toutes les propriétés et les méthodes implémentées sur la classe, nous pouvons le voir comme le "type" d'une classe.

Elle comprend également toutes les interfaces implémentées par la classe.

## L'implémentation d'interface

Lorsqu'une classe implémente une autre classe, cela signifie qu'elle déclare toutes les propriétés définies sur la première classe.

Par exemple :

```
// Prenons une classe :  
class ClasseA {  
    String? prop1;  
  
    ClasseA(this.prop1);  
  
    String method1(String param) => 'Un $param.';  
}  
  
// La classe A a donc comme interface implicite, une propriété d'instance  
// de type String, et une méthode qui prend en paramètre une String.  
  
class ClasseB implements ClasseA {  
    String prop1 = 'une valeur';  
    String? prop2;  
  
    String method1(String param) => 'Retourne autre chose';  
}
```

[Copier](#)

La `ClasseB` implémente la `ClasseA` car elle respecte son API. Elle doit contenir toutes les propriétés et méthodes de la `ClasseA` mais peut en implémenter d'autres, comme par exemple `prop2`.

Il faut également respecter les types des propriétés, les types du retour et des paramètres de la `ClasseA`.

Nous reverrons des cas concrets plus tard.

## Les classes abstraites

Parfois nous voulons définir des interfaces sans les implémenter directement.

Dans ce cas, en Dart, il faut utiliser une **classe abstraite**. Il s'agit en fait d'une classe dont l'interface implicite ne sera implémentée que dans une autre classe.

Les classes abstraites ne peuvent pas être instanciées.

Elles peuvent être implémentées par d'autres classes.

```
abstract class UneClasseAbstraite {  
    String proprieteeAbstraite;  
  
    void methodeAbstraite();  
}  
  
class UneClasseImplementee implements UneClasseAbstraite {  
    String proprieteeAbstraite = 'valeur';  
  
    void methodeAbstraite() => print(proprieteeAbstraite);  
}
```

Nous appelons méthodes abstraites les méthodes définies sur une classe abstraite car elles ne sont pas implémentées : elles ne font rien.

## 14. Import & Export

`import 'Tesla.dart' hide Car;`

on import tout SAUF la classe Car

`import 'Tesla.dart' show Tesla,Car;`

on n'importe rien À L'EXCEPTION  
des classes Tesla et Car

⇒ Librairie chargée seulement lorsque utilisée

`import 'dart:async';`

`import 'Tesla.dart' deferred as Tessy;`

`Future testLibrary() async {`

`await Tessy.loadLibrary();`

`[...]`

`}`

⇒ `export 'cheminDeLExport';`

⇒ Import package natif dart

`import 'dart:packageName';`

Import d'un package téléchargé ds  
le "pub"

`import 'package:flutter/nomPackage';`

main.dart ▶ main

```
1 import 'dart:async';
2
3 main() {
4
5     Future<String> f = Future<String>.delayed(Duration(seconds: 3), (){
6
7         throw Exception('error');
8         // return 'fini';
9     }); // Future.delayed
10
11    f.then((String value){
12        print(value);
13    }).catchError((err){
14        print(err);
15    });
16
17    print('hello');
18
19 }
20
21
```

```
1 import 'dart:async';
2
3 Future getData() {
4     return Future.value('je suis de la data');
5 }
6
7
8 main() async {
9
10    try {
11        var data = await getData();
12        print(data);
13    } catch(e) {
14        print(e);
15    }
16}
17
18
```