

Chapitre 2 : Les bases de Dart pour Flutter

▼ Environnement Dart

Dans ce chapitre nous allons apprendre les bases du langage Dart qui vont nous permettre de coder nos premières applications Flutter.

Exécuter du Dart

Vous avez plusieurs moyens d'exécuter du Dart et nous allons vous les apprendre.

1. Exécuter du Dart localement

Il faut d'abord récupérer le chemin vers sdk de dart dans : `D:\tools\src\flutter\bin\cache\dart-sdk\bin`.

Dans la barre de recherche **Windows** tapez "environnement" ou "env" puis sélectionnez **Modifier Variables d'environnement** puis **Variables d'environnement**, cherchez la variable **Path** puis ajouter le chemin vers le sdk de dart

Ou bien Cliquez sur "Parcourir..." et allez dans le dossier où vous avez extrait Flutter


2. Utiliser le Dart Pad

Une autre manière est d'utiliser le Dart Pad mis à disposition par Google.

Vous pouvez le retrouver ici.

DartPad

An online Dart editor with support for console, web, and Flutter apps

 <https://dartpad.dartlang.org/>

Bases de Dart

En **Dart tout est un objet**, tout ce que vous placez dans une variable est un objet. **Un objet est une instance de la classe Object**, ce qui signifie que tout en **Dart** hérite de la classe **Object**.

Dart est un langage typé comme par exemple **Java** ou **Typescript**. Mais comme pour Typescript, les types peuvent être inférés (c'est-à-dire détectés automatiquement) dans certains cas.

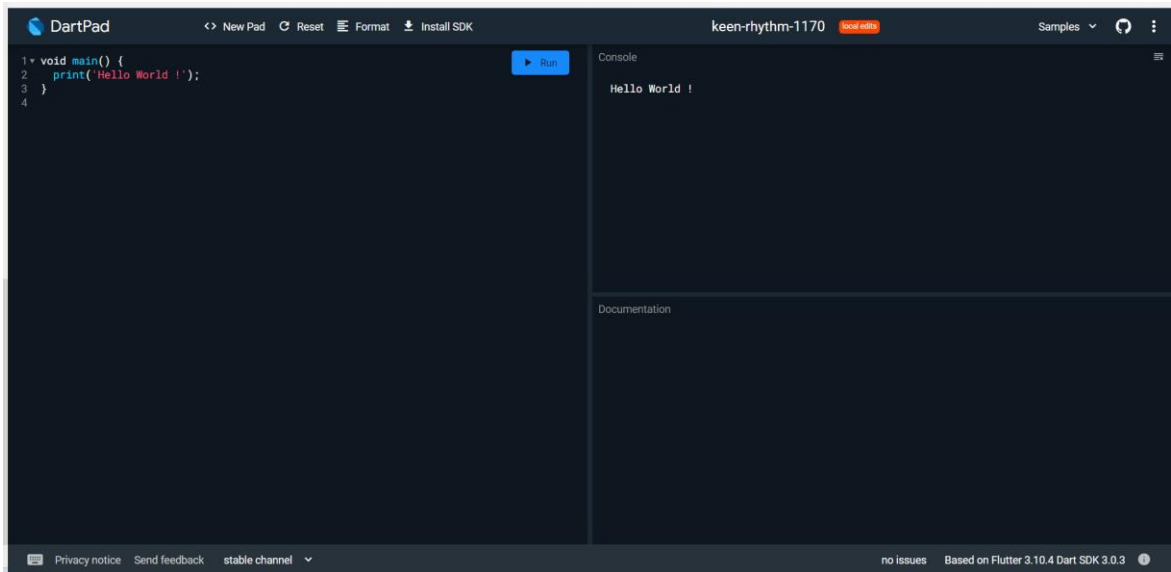
NB: Dart peut être transpilé en JavaScript

Si vous connaissez **Java**, **JavaScript** ou **Typescript**, apprendre les bases de **Dart** sera extrêmement simple et ne nécessitera que ce chapitre !

Nous allons commencer par un petit **Hello World !** comme l'usage nous y oblige ;)

En Dart chaque application doit avoir une fonction `main()` qui sert d'entrée dans l'application.

Voici l'exemple :



▼ Les variables en Dart

Les variables

Les **variables stockent des références à des objets**, et comme nous l'avons vu tout est un objet en **Dart**. Donc une chaîne de caractères est un objet par exemple.

Ce qui est différent du **JavaScript** qui stocke les valeurs pour les types primitifs et des références pour les objets.

var permet de déclarer une variable sans déclarer son type.

Par exemple :

```
var prenom = 'Babacar';
```

Ici, nous avons une variable qui stocke une référence à un objet **String**.

Nous n'avons pas besoin de spécifier le type car il est **inféré** par **Dart** automatiquement.

En revanche, les variables ne sont pas typées dynamiquement en **Dart**, contrairement au **JavaScript**.

Cela signifie que si vous écrivez :

```
var prenom = 'Babacar';
prenom = 27;
```

Vous aurez une erreur car **prenom** a été typé en **String** automatiquement par **Dart**, et son type ne peut pas être modifié ultérieurement.

Définir une variable typée

Nous pourrions cependant typer la variable directement.

Pour cela il faut déclarer un des types **Dart** : **numbers**, **strings**, **booleans**, **lists**, **sets**, **maps**, **runes** et les **symbols**.

1. Numbers (Nombres)

Les nombres en Dart représentent les valeurs numériques. Il existe deux types de nombres : les entiers (**int**) et les décimaux qui sont des nombres flottants(**double**). Les nombres peuvent être utilisés pour effectuer des calculs mathématiques et des opérations arithmétiques.

Exemple :

```
int age = 25;
double prix= 19.99;
```

- Le type `num`

En Dart `num` est un type générique qui représente tous les types numériques, à la fois les entiers et les décimaux. Il peut être utilisé pour stocker des valeurs numériques sans se soucier du type spécifique.

Le type

Voici un exemple d'utilisation du type `num` :

```
num prix= 19.99; // Le type de la variable prix est déterminé automatiquement en fonction de la valeur assignée
num quantite = 5;
num total = prix* quantite ;

print(total); // Affiche : 99.95
```

Dans cet exemple, la variable `prix` est déclarée avec le type `num` et est initialisée avec une valeur décimale. La variable `quantite` est également de type `num` et stocke un entier. En multipliant `prix` par `quantite`, nous obtenons le total, qui est également de type `num`.

`num` est utile lorsque vous voulez travailler avec des valeurs numériques sans vous soucier du type spécifique (int ou double). Cependant, il convient de noter que si vous utilisez le type `num`, certaines opérations spécifiques aux entiers ou aux décimaux peuvent ne pas être disponibles, car elles dépendent du type sous-jacent.

Si vous avez besoin d'opérations spécifiques aux entiers ou aux décimaux, il est recommandé d'utiliser les types `int` ou `double` respectivement, plutôt que le type générique `num`.

2. Strings (Chaînes de caractères)

Les chaînes de caractères en Dart sont utilisées pour représenter du texte. Elles sont entourées de guillemets simples (') ou doubles ("). Les chaînes peuvent contenir des lettres, des chiffres, des symboles et des espaces.

Exemple :

```
String message = "Bonjour, comment ça va ?";
```

3. Booleans (Booléens)

Les booléens en Dart représentent les valeurs logiques. Ils peuvent être soit vrai (true) soit faux (false). Les booléens sont couramment utilisés pour les décisions conditionnelles et les expressions logiques.

Exemple :

```
bool isActive = true;
bool isLoggedIn = false;
```

4. Lists (Listes)

Les listes en Dart sont des collections ordonnées d'éléments. Elles peuvent contenir différents types de données et être de taille variable. Les éléments d'une liste sont accessibles par leur indice, qui commence à 0.

Exemple :

```
List<int> numbers = [1, 2, 3, 4, 5];
List<String> names = ['Alice', 'Bob', 'Charlie'];
```

5. Sets (Ensembles)

Les ensembles en Dart sont des collections non ordonnées d'éléments uniques. Ils ne permettent pas d'avoir des doublons. Les ensembles sont utiles lorsque vous avez besoin de vérifier rapidement si un élément est présent ou non.

Exemple :

```
Set<int> uniqueNumbers = {1, 2, 3, 4, 5};
```

6. Maps (Dictionnaires)

Les dictionnaires en Dart, également appelés maps, sont des collections d'associations clé-valeur. Chaque élément est composé d'une clé unique et d'une valeur correspondante. Les clés doivent être uniques, mais les valeurs peuvent être dupliquées.

Exemple :

```
Map<String, String> capitals = {  
  'Senegal': 'Dakar',  
  'United States': 'Washington D.C.',  
  'Japan': 'Tokyo'  
};
```

7. Runes

Les runes en Dart représentent un point de code Unicode. Ils sont utilisés pour manipuler des caractères Unicode spécifiques, tels que des emojis ou des caractères internationaux.

Exemple :

```
Rune heart = '\u2764'.runes.first;  
print(String.fromCharCode(heart)); // Affiche : ❤️
```

8. Symbols (Symboles)

Les symboles en Dart sont utilisés pour représenter des identifiants uniques. Ils sont souvent utilisés dans les métaprogrammes ou les réflexions pour identifier des éléments tels que des fonctions, des classes ou des variables.

Exemple :

```
Symbol methodName = Symbol('myMethod');
```

Ces types de données en Dart sont utilisés pour représenter et manipuler différentes valeurs dans vos programmes. En comprenant ces types, vous pouvez créer des variables et des structures de données appropriées pour votre application.

Interpolation de variable

Comme en **JavaScript**, vous pouvez interpoler des variables avec **\$var** ou **\${var}** :



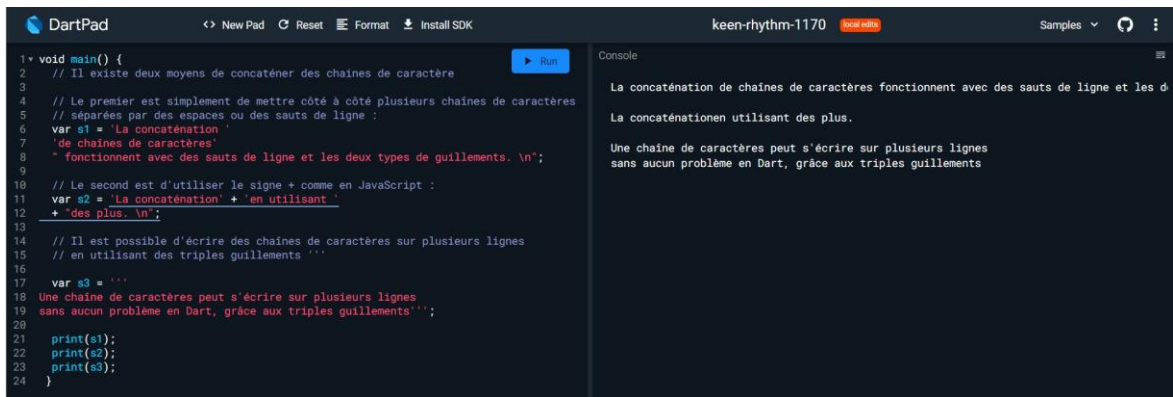
```
1 void main() {  
2   int x = 100;  
3   String s = 'Bonjour ! Je suis une chaîne. Vous pouvez m\'échapper !';  
4  
5   // Interpolation  
6   String interpolation1 = 'La valeur de x est $x';  
7   String interpolation2 = 'La valeur de s en lowercase est ${s.toLowerCase()}';  
8  
9   print(interpolation1);  
10  print(interpolation2);  
11 }  
12  
13  
14
```

Console

```
La valeur de x est 100  
La valeur de s en lowercase est bonjour ! je suis une chaîne. vous pouvez m'échapper !
```

Concaténation de chaînes de caractères et les chaînes multilignes

La concaténation et l'écriture sur plusieurs lignes de chaînes de caractères est extrêmement pratiques en **Dart** :



Les constantes

En **Dart** il existe deux types de constantes : **const** et **final**.

Les **const** sont des constantes qui doivent être définies au moment de la compilation. Elles sont initialisées et assignées au même moment.

Les **final** sont des constantes, qui ne peuvent donc être initialisées qu'une seule fois, mais qui peuvent être assignées pendant l'exécution. Autrement dit, elles peuvent recevoir une valeur déterminée au moment de l'exécution.

```
void main() {  
  // Deux moyens de déclarer des const :  
  const pi = 3.14;  
  const double pi2 = 3.13;  
  
  // Ne compilera pas :  
  pi = 24;  
  
  // Même chose pour les finals :  
  final x = 1;  
  final int y = 2;  
  
  // La différence entre les deux :  
  const t = DateTime.now(); // ne compilera pas !  
  final t = DateTime.now(); // valeur connue lors de l'exécution mais initialisation fixe  
}
```

Il y a quelques erreurs dans le code fourni :

1. Les constantes (const) ne peuvent pas être réassignées une fois qu'elles ont été déclarées. Par conséquent, la ligne `pi = 24;` entraînera une erreur de compilation. Pour corriger cela, vous devez soit utiliser `final` au lieu de `const`, soit choisir une nouvelle valeur lors de la déclaration initiale.
2. Dans la ligne `const t = DateTime.now();`, vous essayez de déclarer une constante en utilisant `DateTime.now()`, qui est une valeur dynamique et ne peut pas être évaluée au moment de la compilation. Par conséquent, cela entraînera également une erreur de compilation. Si vous avez besoin d'une valeur constante basée sur l'heure actuelle, vous devrez la déclarer comme `final` plutôt que `const`.

▼ Les lists et les map

1. Les lists

Les listes sont comme des tableaux en **JavaScript**.

Définir une list

Pour définir une liste vous pouvez utiliser **var** et des crochets `[]` :

```
var maListe = [1, 2, 3];
```

Ici, **Dart** va inférer que le type est `List<int>`, c'est-à-dire qu'il va attribuer automatiquement le type List contenant des `int`.

Il ne sera donc pas possible d'ajouter autre chose :

```
var maListe = [1,2,3];

maListe.add('Hello !'); // Error: The argument type 'String'
// can't be assigned to the parameter type 'int'.
```

Vous pouvez également définir explicitement une **list** et la typer comme vous le souhaitez :

```
List<int> maListe = [1,2,3];
```

La propriété **length**

Comme en **JavaScript**, les **lists** commencent à l'**index 0**.

Les **lists** ont une propriété **length** pour connaître la longueur de celle-ci.

Le dernier élément d'une **list** est donc à l'**index list.length - 1**.

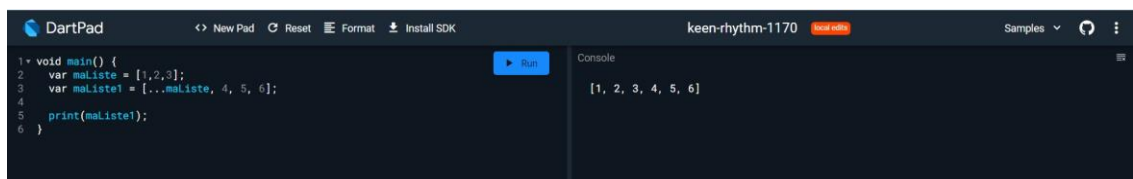
Accéder à un élément d'une **list**

Pour accéder à un élément d'une list, il suffit de faire **list[index]**.

```
var list = [1, 2, 3];
list[1]; // 2
```

L'opérateur **spread (...)**

Comme en **JavaScript** depuis **ES6**, Dart permet d'utiliser l'opérateur **spread** pour copier facilement une **list** dans une autre **list** :



Quelques méthodes

Il existe environ 50 méthodes en **Dart** pour les **lists**. Nous les verrons au fur à mesure dès que nous en aurons besoin.

Mais nous allons voir quelques méthodes basiques dès maintenant pour les **lists** : **add()**, **addAll()**, **clear()**, **indexOf()**, **contains()**, **remove()** et **removeAt()**.

```
void main() {
  var list = [1,2,1];

  // Ajouter une valeur à la fin de la List :
  list.add(2);

  // Enlever la première occurrence d'une valeur spécifique d'une List :
  // Retourne true si un élément a été enlevé, false sinon.
```

```

list.remove(1);
print(list);

// Enlever l'élément à l'index spécifié :
list.removeAt(0);
print(list);

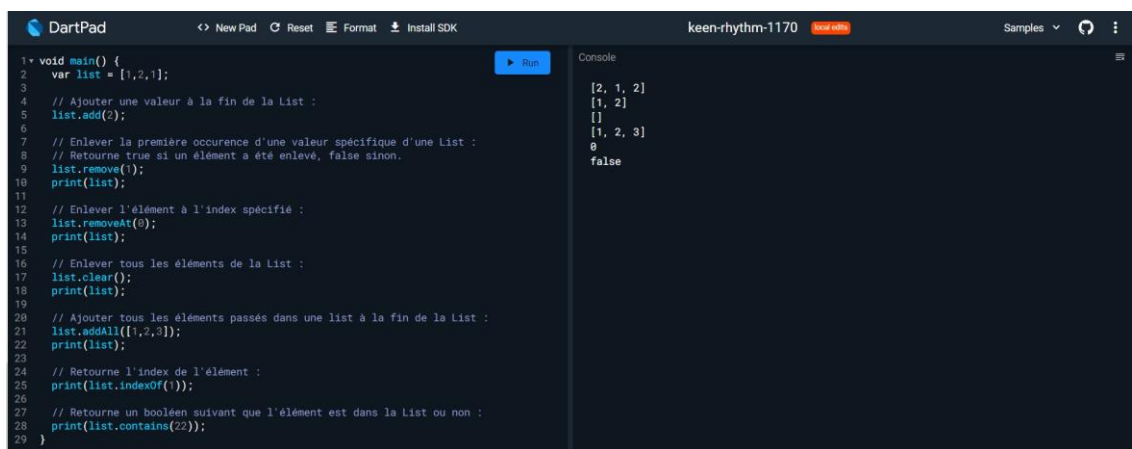
// Enlever tous les éléments de la List :
list.clear();
print(list);

// Ajouter tous les éléments passés dans une list à la fin de la List :
list.addAll([1,2,3]);
print(list);

// Retourne l'index de l'élément :
print(list.indexOf(1));

// Retourne un booléen suivant que l'élément est dans la List ou non :
print(list.contains(22));
}

```



2. Les maps

Un **map** est un objet qui associe des clés et des valeurs. Les clés et les valeurs peuvent avoir tout type d'objet.

Les clés doivent être uniques mais pas les valeurs.

En fait, les **maps** en **Dart** sont extrêmement semblables aux **maps** en **JavaScript**. Il n'existe par contre pas d'objets littéraux comme en **JavaScript**.

Définir un map

Vous pouvez définir un **map** avec **var** :

```

var map = {
  'clé1': 'valeur1',
  'clé2': 'valeur2'
};

```

Vous pouvez également typer un map :

```

Map<String, String> map1 = {'clé1': 'valeur1'};
Map<int, String> map2 = {1: 'valeur1'};
Map<String, double> map3 = {'clé3': 45.22};

```

Accéder à un élément d'un map

Vous pouvez simplement utiliser **map[cle]** :

```

Map<String, String> map1 = {'cl  1': 'valeur1'};
Map<int, String> map2 = {1: 'valeur1'};
Map<String, double> map3 = {'cl  3': 45.22};

print(map1['cl  1']); // 'valeur1'
print(map2[1]); // 'valeur1'
print(map3['cl  3']); // 45.22

```

La propri  t   length

Les **maps** ont   galement une propri  t   **length** qui fonctionne de la m  me mani  re que pour les **lists**.

Quelques autres propri  t  s

Nous allons voir les propri  t  s **values**, **keys** et **entries** :

```

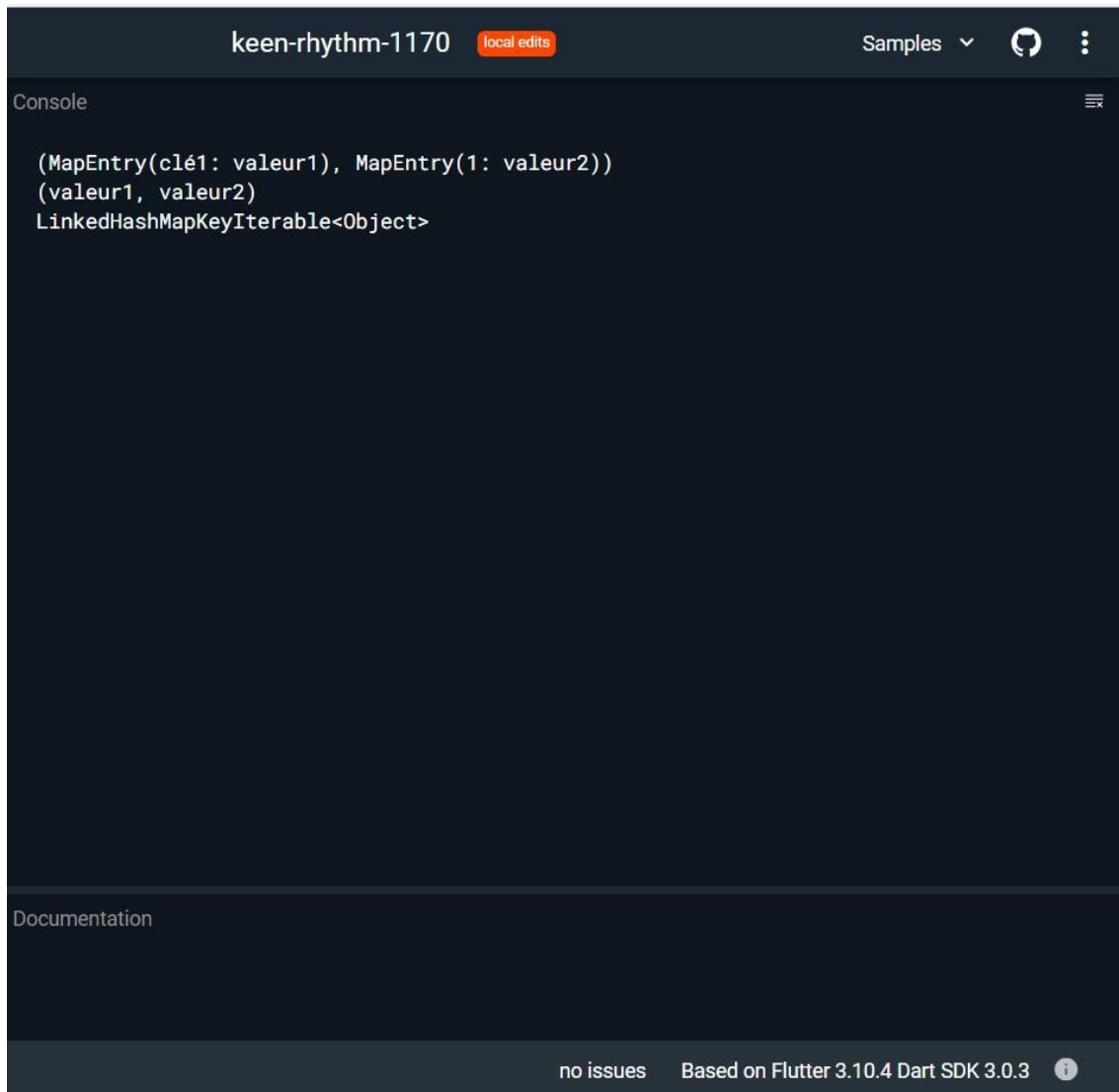
var map1 = {
  'cl  1': 'valeur1',
  1: 'valeur2'
};

// La propri  t   entries permet d'obtenir les paires cl  /valeur :
print(map1.entries);

// La propri  t   values permet d'obtenir les valeurs :
print(map1.values);

// La propri  t   keys permet d'obtenir les cl  s :
print(map1.keys.runtimeType);

```

Opérateur spread

Il est possible d'utiliser l'opérateur **spread** avec un **map** également :

```
void main() {  
  var map1 = {  
    'clé1': 'valeur1',  
    1: 'valeur2'  
  };  
  
  var map2 = {  
    ...map1,  
    'clé3': 44.23  
  };  
  
  print(map2); //(clé1: valeur1, 1: valeur2, clé3: 44.23)  
}
```

Quelques méthodes

Nous allons voir quelques méthodes basiques pour les **maps** : **containsKey**, **containsValue** et **remove** :

```
void main() {  
  var map1 = {  
    'clé1': 'valeur1',  
    1: 'valeur2'  
  };  
}
```

```

};

// containsKey() permet de savoir si une clé existe :
print(map1.containsKey('clé1')); //true

// containsValue() permet de savoir si une valeur existe :
print(map1.containsValue(1)); //false

// remove() permet de supprimer une paire clé/valeur :
map1.remove('clé1');
print(map1); // {1: valeur2}
}

```

Bien sûr nous sommes loin d'avoir fait le tour des **lists** et des **maps** en **Dart** ! Mais c'est une approche suffisante pour commencer **Flutter**.

▼ Les fonctions

Les fonctions en Dart

Comme le reste en **Dart**, les fonctions sont des objets.

Elles peuvent donc être assignées à des variables et passées comme arguments à d'autres fonctions.

En **Dart** il est recommandé de **typer le retour des fonctions**, par exemple :

```

void direBonjour() {
  print('Bonjour');
}
direBonjour();

```

Ici la fonction ne retourne rien car **print()** retourne **void**.

Une fonction doit toujours **return** quelque chose, si nous n'utilisons pas **return**, alors **return null**; est ajouté par défaut à la fonction.

Vous pouvez omettre le type retourné et cela fonctionnera, mais ce n'est pas recommandé.

Les fonctions fléchées

En **Dart** il existe aussi des fonctions fléchées qui fonctionnent exactement comme en JavaScript:

=> expression équivaut à **{ return expr; }**.

Par exemple :

```

void direBonjour() => print('Bonjour');
direBonjour();

```

Les paramètres

En **Dart** il vaut mieux également typer les paramètres.

1. Typer les paramètres

Nous allons prendre un premier exemple :

```

void main() {
  var nomComplet = getFullName('Babacar', 'NDIAYE');
  print(nomComplet);
}

String getFullName(String prenom, String nom) {
  return "$prenom $nom";
}

```

```
//Console
Babacar NDIAYE
```

Ce programme déclare une fonction `getFullName` qui prend deux paramètres : `prenom` et `nom` de type `String`. Elle retourne une chaîne de caractères contenant la combinaison du prénom et du nom séparés par un espace.

2. Paramètres nommés et null safety

Jusqu'à maintenant nous avons vu comment utiliser des **paramètres positionnels**.

En **Dart**, il est également possible d'utiliser des **paramètres nommés** pour cela il suffit de faire lors de la définition :

```
nomFonction({param1, param2}) {}
```

Et lors de l'utilisation :

```
nomFonction(param1: val1, param2: val2);
```

Les **paramètres nommés** offrent plus de **flexibilité** lors de l'appel de fonctions en spécifiant **explicitement** les noms des paramètres avec leurs **valeurs correspondantes**.

NB: Les paramètres nommés sont optionnels sauf si vous utilisez `required` devant le nom du paramètre.

Voici un exemple :

```
void main() {
  var nomComplet = getFullName(nom: 'NDIAYE');
  print(nomComplet);
}

String getFullName({required String prenom, required String nom}) {
  return "$prenom $nom";
}

//Console
Error: Required named parameter 'prenom' must be provided.
```

Lors de la compilation de ce programme on aura une erreur puisque le paramètre prénom est obligatoire.

Si vous utilisez des paramètres nommés et que vous ne fournissez pas une valeur par défaut avec `=` (nous le verrons plus bas dans la leçon) ou que vous ne spécifiez pas qu'ils sont obligatoires avec **required**, vous aurez une erreur.

En effet, dans ce cas, leur valeur par défaut est **null** et donc il faut le préciser en ajoutant un `?` à la fin du type.

Le point d'interrogation à la fin d'un type permet d'ajouter la valeur **null** à ce type.

En effet, Dart a une fonctionnalité appelée **null safety (sécurité contre les null)** qui vous oblige à spécifier si une variable ou un paramètre peut contenir la valeur **null** ou non.

Par défaut, Dart va considérer qu'il n'est pas possible que la valeur soit **null**. Il faut donc utiliser le point d'interrogation pour le permettre.

Voici un exemple :

```
void main() {
  var nomComplet = getFullName(prenom: 'Babacar', nom: 'NDIAYE');
  print(nomComplet);
}

String getFullName({String? prenom, String? nom}) {
  return "$prenom $nom";
}
```

3. Paramètres optionnels ou requis

En **Dart**, les paramètres peuvent être optionnels ou obligatoires.

Par défaut les paramètres positionnels sont obligatoires :

```
void main() {
  var nomCompleet = getFullName('Babacar'); // Error: Too few positional arguments:
  // 2 required, 1 given.

  print(nomCompleet);
}

String getFullName(String prenom, String nom) {
  return "$prenom $nom";
}
```

Pour rendre un paramètre optionnel il faut utiliser **[param]** :

```
void main() {
  var nomCompleet = getFullName('Babacar');
  print(nomCompleet);
}

String getFullName(String prenom, [String? nom]) {
  return "$prenom $nom";
}

//Console
Babacar null
```

4. Paramètres par défaut

Il est également possible de définir la valeur par défaut des paramètres en **Dart**.

Pour cela il suffit de faire : **type param = valeurParDefaut**.

Comme précisé plus haut, lorsqu'un paramètre est optionnel, il faut soit lui donner une valeur par défaut, soit préciser que la valeur peut être **null** en ajoutant un **?** à la fin du type :

```
void main() {
  var nomCompleet = getFullName();
  print(nomCompleet);
}

// Fonction avec deux paramètres optionnels et dont
// le premier paramètre a une valeur par défaut :
String getFullName([String prenom = 'Inconnu', String? nom]) {
  return "$prenom $nom";
}

//Console
Inconnu null
```

Même chose avec les paramètres nommés :

```
void main() {
  var nomCompleet = getFullName(prenom: 'Babacar');
  print(nomCompleet);
}

// Fonction avec deux paramètres optionnels et dont
// le premier paramètre a une valeur par défaut :
String getFullName({String? prenom, String nom = 'Inconnu'}) {
  return "$prenom $nom";
}

//Console
Babacar Inconnu
```

Les fonctions anonymes

Comme en **JavaScript**, il est possible d'utiliser des **fonctions anonymes** appelées également **lambdas** ou **fonctions fléchées**:

```
[[Type] param1[, ...]] {  
  codeBlock;  
};
```

Par exemple :

```
void main() {  
  List<int> list = [1,2,3,4];  
  
  list.forEach((item) {  
    print(item);  
  });  
  print('\nFonction fléchée anonyme\n');  
  // Ou encore avec une fonction fléchée anonyme :  
  list.forEach((item) => print(item));  
}  
  
//Console  
1  
2  
3  
4  
  
Fonction fléchée anonyme  
  
1  
2  
3  
4
```

Attention ! Contrairement au **JavaScript**, si vous n'avez qu'un seul paramètre dans une fonction fléchée anonyme vous ne pouvez pas omettre les parenthèses.

Voici un exemple pour clarifier cela :

```
void main() {  
  var numbers = [1, 2, 3, 4, 5];  
  
  // Utilisation d'une fonction fléchée anonyme avec un seul paramètre  
  var squares = numbers.map((number) => number * number);  
  
  print(squares.toList()); // Affiche : [1, 4, 9, 16, 25]  
}
```

Dans cet exemple, nous utilisons une fonction fléchée anonyme avec un seul paramètre `number` pour calculer le carré de chaque nombre dans la liste `numbers`. La syntaxe `(number) => number * number` est utilisée pour définir la fonction fléchée anonyme.

Notez que les parenthèses autour du paramètre `number` sont nécessaires. Si vous essayez de les omettre en utilisant simplement `number => number * number`, cela entraînera une erreur de syntaxe.

Donc, pour récapituler, lorsque vous utilisez une fonction fléchée anonyme en Dart avec un seul paramètre, assurez-vous d'inclure les parenthèses autour du paramètre. Cela garantit que votre code est correctement interprété par le compilateur Dart.

Passer des fonctions comme argument à d'autres fonctions

Comme en **JavaScript**, les fonctions peuvent être passées en argument à d'autre fonction.

Nous pouvons donc écrire :

```
main() {  
  List<int> list = [1,2,3,4];  
  
  list.forEach(print);  
}
```

Voici un autre exemple avec une fonction fléchée que nous définissons :

```
void main() {
  List<String> list = ['je', 'suis', 'en', 'lowercase'];

  var listUpper = list.map(toUpperCase);
  print(listUpper);
}

String toUpperCase(String elem) => elem.toUpperCase();

//Console
(JE, SUIS, EN, LOWERCASE)
```

```
void main() {
  var result = calculate(5, 3, (a, b) => a + b);
  print(result);
}

int calculate(int a, int b, int Function(int, int) operation) {
  return operation(a, b);
}

//Console
8
```

Dans cet exemple, nous avons une fonction anonyme passée comme argument à la fonction `calculate`. La fonction anonyme prend deux paramètres `a` et `b`, et retourne la somme des deux nombres.

La fonction `calculate` elle-même prend trois paramètres : `a`, `b` et `operation`. `operation` est une fonction qui attend deux entiers et retourne un entier. Dans notre cas, nous passons une fonction anonyme `(a, b) => a + b` comme `operation`, qui effectue simplement l'addition des deux nombres.

Lorsque nous appelons `calculate(5, 3, (a, b) => a + b)`, la fonction `calculate` exécute l'opération fournie (dans notre cas, l'addition) en utilisant les valeurs `5` et `3` comme arguments, et retourne le résultat `8`. Ce résultat est ensuite stocké dans la variable `result` et affiché à l'aide de la fonction `print`.

Les fonctions anonymes en Dart sont utiles lorsque vous souhaitez définir des fonctionnalités spécifiques à un certain contexte ou lorsque vous souhaitez passer des fonctions en tant qu'arguments à d'autres fonctions. Elles offrent une flexibilité et une expressivité accrues dans la manière dont vous définissez et utilisez les fonctions dans votre code.

▼ Les structures de contrôle

Les blocs conditionnels

Comme dans la plupart des langages nous pouvons utiliser `if`, `else` et `else if` :

```
void main() {
  bool test1 = false;
  bool test2 = true;

  if (test1) {
    print('Par là !');
  } else if (test2) {
    print('Ici !');
  } else {
    print('Là !');
  }
}
```

A noter qu'il n'y a pas de **coercition** contrairement au **JavaScript**. Cela signifie qu'une chaîne de caractères ne sera pas transformée en true par exemple, dans un contexte de test.

Vous ne pouvez donc pas écrire :

```
void main() {
  String test1 = 'J'existe !';
  bool test2 = true;

  if (test1) { // Error: A value of type 'String' can't be assigned to
// a variable of type 'bool'.
    print('Par là !');
  } else if (test2) {
    print('Ici !');
  } else {
    print('Là !');
  }
}
```

Vous devez écrire :

```
void main() {
  String test1 = 'J'existe !';
  bool test2 = true;

  if (test1 != null) {
    print('Par là !'); // Par là !
  } else if (test2) {
    print('Ici !');
  } else {
    print('Là !');
  }
}
```

L'opérateur ternaire

Comme en **JavaScript** vous pouvez utiliser des ternaires.

```
void main() {
  int age = 17;

  bool isMajeur = age >= 18 ? true : false;
  print(isMajeur); //false
}
```

Les boucles

En **Dart**, nous pouvons utiliser les boucles **for**, **while** et **do/while**.*

• Les boucles for

Les boucles **for** fonctionnent pareil qu'en **JavaScript** :

```
void main() {
  for (int i = 0; i < 5; i++) {
    print('Hello ${i + 1}');
  }
}
```

Vous pouvez utiliser également **forEach** sur des itérables comme les **List**, mais vous ne pouvez pas accéder à l'index de l'itération en cours :

```
void main() {
  List<int> liste = [1, 2, 3];
  int result = 0;

  liste.forEach((elem) => result = result + elem);
  print(result);
}
```

Vous pouvez également utiliser `for ... in` :

```
void main() {
  List<int> liste = [1, 2, 3];
  int result = 0;

  for (var e in liste) {
    result = result + e;
  }

  print(result);
}
```

- **Les boucles while**

Les boucles `while()` permettent d'exécuter un bloc d'instructions tant qu'une condition n'est pas remplie :

```
void main() {
  List<int> liste = [1, 2, 3];

  while (liste.isNotEmpty) { //liste.length != 0
    print(liste);
    liste.removeLast();
  }

  print(liste);
}
//Console
[1, 2, 3]
[1, 2]
[1]
[]
```

- **Les boucles do / while**

Avec les boucles `do / while`, la condition est évaluée après la boucle :

```
void main() {
  List<int> liste = [1, 2, 3];

  do {
    print(liste);
    liste.removeLast();
  } while (liste.length != 0);

  print(liste);
}
//Console
[1, 2, 3]
[1, 2]
[1]
[]
```

- **Utilisation de break et continue**

break permet d'arrêter la boucle si une condition survient..

continue permet de sauter une itération suivant une condition.


```

void main() {
    List<int> liste = [1, 2, 3];

    print("break");
    while (liste.length != 0) {
        // Si la liste n'a plus qu'un élément nous stoppons la boucle :
        if (liste.length == 1) break;
        print(liste);
        liste.removeLast();
    }
    print("continue");
    for (var i = 2; i < 10; i++) {
        // Nous sautons l'itération si i est pair :
        if (i % 2 == 0) continue;
        // Donc seuls les éléments impairs sont ajoutés :
        liste.add(i);
    }

    print(liste);
}
//Console
break
[1, 2, 3]
[1, 2]
continue
[1, 3, 5, 7, 9]

```

Les switch / case

Un **switch / case** permet de comparer une valeur à un **int** ou à un **string**.

```

void main() {
    const int a = 3;
    switch (a) {
        case 1:
            print('Ici');
            break;
        case 2:
            print('là');
            break;
        default:
            print('LA !');
            break;
    }
}
//Console
LA !

```