



DEVELOPPEMENT D'APPLICATION MOBILE

Chapitre 2 : Découverte du Langage Dart

Sommaire

1. [Introduction](#)
2. [Prise en main du CLI de Dart](#)
3. [TP 1: Premier programme Dart](#)
4. [Les commentaires](#)
5. [Déclaration de variables](#)
6. [Les variables statiques simples](#)
7. [Les variables statiques complexes](#)
8. [Les instructions *var*, *final* et *const*](#)
9. [Les instructions *late* et *dynamic*](#)
10. [Les tests](#)
11. [Les boucles](#)
12. [Les fonctions et procédures](#)
13. [Les énumérations](#)
14. [Les importations](#)
15. [L'asynchrone](#)
16. [Les exceptions](#)
17. [La programmation orientée objet](#)

Introduction



Créé en 2011 par Google, Dart est un langage de programmation utilisé pour le développement d'applications côté client (web, mobile et bureau) avec comme principales caractéristiques:

- Programmation Orientée Objet (POO) comme les langages Java, C#,...
- **Typage explicite** (statique) comme dans les langages typés (C, C++, ...) et **typage implicite** (dynamique) comme dans les langages non typés (Python, Javascript).
- **Nulle safety** par défaut (une variable en Dart doit contenir obligatoirement une valeur sauf si on la déclare explicitement nulle).
- Compilation JIT (Just In Time) avec une interprétation lors de l'exécution (Hot Reload dans Flutter) ce qui permet de visualiser instantanément les changements.
- Compilation AOT (Ahead Of Time) avec une compilation en code machine ou en Javascript.
- Concurrency indépendante avec les isolates (unités d'exécution indépendantes qui ne partagent pas la mémoire) contrairement à d'autres langages qui utilisent les Thread pour la concurrence.
- Intègre un ramasse-miettes (garbage collector) qui permet de gérer la mémoire de manière automatique et efficace.
- Possibilité de développer des applications mobiles (avec Flutter), applications Web, applications Serveur, applications CLI (Command-Line Applications).

Pour plus d'informations visiter [Site de Dart](#), [Blog Medium de Dart](#), [Liste de diffusion de Dart](#).

Prise en main du CLI de Dart

Pour exécuter du code Dart on peut soit utiliser le SDK (Software Development Kit) sur une machine [Installation de Dart](#) ou utiliser [DartPad](#) pour une exécution en ligne avec un navigateur Web.

Le code en Dart doit être écrit dans des fichiers d'extension **.dart**

Pour exécuter un code Dart, exécuter la commande ***dart NomDuFichier.dart***

Pour vérifier la version de Dart installée sur une machine, exécuter la commande: ***dart --version***

Pour afficher la liste des commandes possibles avec Dart, exécuter ***dart -h*** ou ***dart --help***

Pour afficher la documentation sur une commande spécifique exécuter la commande au format:

dart {commande} -h ou ***dart {commande} --help***

Par exemple pour afficher la documentation de la commande ***create*** on exécutera: ***dart create --help***

Pour afficher un message dans la console, on utilisera dans le code ***print("message à afficher");***

Chaque instruction en Dart doit se terminer par un point-virgule.

TP 1: Premier programme Dart

1. Installer le SDK Dart
2. Installer l'extension Dart de VSCode
3. Quelle est la version de Dart installée ?
4. Créer avec DartPad un programme qui affiche le message « Hello, ce programme a été écrit en Dart ! »
5. Créer un projet Dart avec VSCode.
6. Créer un projet Dart en ligne de commande avec pour nom tp1-premier-programme-dart.
7. Explorer l'arborescence du projet.
8. Dans le projet tp1-premier-programme-dart écrire un programme qui affiche le message « Hello, ce programme a été écrit en Dart ! ».
9. Exécuter le programme en ligne de commande.
10. Compiler le programme en Javascript.
11. Exécuter la version en Javascript et comparer le résultat avec celui obtenu en Dart.

Les commentaires

On dispose de trois types de commentaires en Dart, les commentaires sur une ligne, les commentaires multilignes et les commentaires de documentation.

```
1 /// Ce commentaire est un commentaire de documentation.  
2 /// Il doit être utilisé pour documenter les librairies, classes....  
3  
4 library;  
5  
6 /*  
7  * Ce commentaire est un commentaire multilignes  
8  * Il doit être encadré de /* ... */  
9  */  
10 void main() {  
11     // Ce commentaire est un commentaire sur une seule ligne  
12     print("Premier programme en Dart !");  
13 }
```



Premier programme en Dart !

Déclaration de variables

La déclaration d'une variable en Dart respecte le format suivant:

type_variable nom_variable = valeur_variable;

Dart a deux caractéristiques principales qui le différencie des autres langages:

- Il a le **null safety** ce qui signifie que toute variable déclarée doit avoir une valeur jamais **nulle** sauf si on spécifie à la déclaration que la variable peut être **nulle**.
- Une variable peut avoir un typage statique ou dynamique.

La déclaration d'une variable qui peut être nulle a pour format (ajout du point d'interrogation à la fin du type):

type_variable? nom_variable = valeur_variable;

La déclaration d'une variable qui peut avoir différentes valeurs de types différents (utilisation de **dynamic** comme type):

dynamic nom_variable = valeur_variable;

Une variable de type **dynamic** peut à un moment donné contenir un entier et après une liste ou une valeur nulle. C'est une variable comparable à celle utilisée dans les langages non typés comme Python.

Les variables statiques simples

Les variables statiques peuvent avoir les types simples suivants:

- ***int*** et ***double*** pour les nombres.
- ***String*** pour les chaînes de caractères.
- ***bool*** pour les booléens.

Variables non nulles

```
int nbre = 1;  
double moyenne = 12.1;  
bool estMajeur = True;  
String nom = "John";
```

Variables pouvant être nulles

```
int? nbre = 1;  
double? moyenne;  
bool? estMajeur = True;  
String? nom = "John";
```

Quand une variable est déclarée avec désactivation du nulle safety (? à son type) alors cette variable peut ne pas avoir de valeur lors de sa déclaration elle est donc initialisée à null.

Les variables statiques complexes

Les variables statiques peuvent avoir les types complexes suivants:

- **List** pour les listes et tableaux.
- **Set** pour les collections non ordonnées et avec des valeurs uniques.
- **Map** pour les objets associant une valeur à une clé.

Il existe d'autres types complexes comme les Records, Runes, Symbols moins fréquents voir [Types intégrés de Dart](#).

Variables non nulles

```
List<int> nbres = [1, 2, 3];  
Set<int> nbres = {1, 2, 3};  
Map<String, int> personnes = {  
  "john": 18,  
  "doé": 19  
};
```

Variables pouvant contenir nulle

```
List<int?> nbres = [1, 2, null];  
Set<int?> nbres = {1, null, 3};  
Map<String?, int?> personnes = {  
  "john": 18,  
  "doé": null,  
  null: 15  
};
```

Ne pas écrire Set? Ou List? Ou Map?, on ne peut pas créer un type complexe en désactivant le nulle safety sur le type de base.

Les instructions *var*, *final* et *const*

L'instruction *var* permet de déclarer une variable dont le type sera déterminé à partir de celui de sa valeur d'initialisation.

```
1 void main() {  
2   var nbre = 1;  
3   // Affichera int car le type de la variable nbre est initialisé au type de sa valeur d'initialisation  
4   print(nbre.runtimeType);  
5 }
```

On ne peut attribuer une valeur d'autres types après initialisation de la variable.

On utilisera les instructions *const* et *final* pour déclarer des variables dont les valeurs ne changent pas après initialisation.

```
1 void main() {  
2   final int nbre = 1;  
3   print(nbre);  
4   final double moyenne = 12.0;  
5   print(moyenne);  
6   final String name = "Oro";  
7   print(name);  
8   const names = ["Oro", "Iri", "Rio"];  
9   print(names);  
10  final Map<String, int> user = const {"age": 12, "moyenne": 15};  
11  print(user);  
12 }
```

Il existe une différence entre *const* (constante lors de la compilation) et *final* (constante lors de l'exécution). Voir [const et final en Dart](#).

Les instructions *late* et *dynamic*

Par défaut chaque variable déclarée en Dart doit avoir une valeur sauf si elle est déclarée comme pouvant être nulle.

L'instruction *late* sera utilisée pour déclarer une variable dont l'initialisation sera effectuée après la déclaration. On l'utilisera assez fréquemment en Flutter.

```
1 int age;  
2  
3 void main() {  
4   age = 12;  
5   print(age);  
6 }
```

compileDDC
main.dart:1:5: Error: Field 'age' should be initialized because its type 'int' doesn't allow null.
int age;
 ^^^

Pour éviter l'erreur ci-dessus, on va utiliser l'instruction *late*.

```
1 late int age;  
2  
3 void main() {  
4   age = 12;  
5   print(age);  
6 }
```

12

L'instruction *dynamic* permet de déclarer une variable pouvant prendre des valeurs de types différents.

```
1 void main() {  
2   dynamic nameOrAge = "toto";  
3   print(nameOrAge);  
4   print(nameOrAge.runtimeType);  
5   nameOrAge = 12;  
6   print(nameOrAge);  
7   print(nameOrAge.runtimeType);  
8 }
```

toto
String
12
int

Les tests

Pour exécuter les tests avec ou sans alternative, on utilisera les instruction *if*, *else* ou *else if*.

```
1 // Programme qui détermine si un nombre est positif ou négatif ou nul
2 int nbre = -1;
3
4 void main() {
5     if (nbre > 0) {
6         print("Le nombre $nbre est positif.");
7     } else if (nbre < 0) {
8         print("Le nombre $nbre est négatif.");
9     } else {
10        print("Le nombre $nbre est nul");
11    }
12 }
```

Le nombre -1 est négatif.

```
1 // Programme qui détermine si un nombre est positif ou négatif ou nul
2 int nbre = 0;
3
4 void main() {
5     if (nbre > 0) {
6         print("Le nombre $nbre est positif.");
7     } else if (nbre < 0) {
8         print("Le nombre $nbre est négatif.");
9     } else {
10        print("Le nombre $nbre est nul.");
11    }
12 }
```

Le nombre 0 est nul.

Exercice: Ecrire un programme qui analyse deux nombres et affiche le signe de leur produit négatif ou positif ou nul.

Les tests

On peut utiliser l'instruction *switch* pour tester la valeur d'une variable avec une série de valeurs ou conditions.

Syntaxe de switch

```
switch(variable) {  
    case valeur1 ou condition:  
        code_à_exécuter_si_condition_sur_valeur1  
        break;  
    case valeur2 ou condition:  
        code_à_exécuter_si_condition_sur_valeur2  
        break;  
    default:  
        code_à_exécuter_par_défaut  
        break;  
}
```



```
1 // Programme qui détermine si un nombre est positif ou négatif ou nul avec switch  
2 final int nbre = 12;  
3  
4 void main() {  
5     switch(nbre) {  
6         case > 0:  
7             print("Le nombre $nbre est positif.");  
8             break;  
9         case < 0:  
10            print("Le nombre $nbre est négatif.");  
11            break;  
12        default:  
13            print("Le nombre $nbre est nul.");  
14            break;  
15    }  
16 }
```

L'instruction *break* est importante quand on n'utilise pas de return dans le *case*. Voir [Switch](#) ou [if/else](#).

Exercice: Ecrire un programme qui affiche la tranche d'âge en fonction de la catégorie Poussin (de 6 à 7 ans), Poupille (de 8 à 9 ans), Minime (de 10 à 11 ans) et Cadet (après 12 ans).

Les boucles

Pour effectuer des boucles conditionnelles (ou indéfinies) on utilisera l'instruction **while** et **for** pour les boucles inconditionnelles (ou avec compteur ou définies).

Syntaxe de while

```
while(condition) {  
    code à répéter tant que la condition est vraie  
}
```

Syntaxe de for

```
for(départ; fin; incrément) {  
    code à répéter tant que la condition est vraie  
}
```

```
1 // Affichage des nombres de 0 à 10 avec while  
2 int nbre = 0;  
3  
Run | Debug  
4 void main() {  
5     while (nbre <= 10) {  
6         print(nbre);  
7         nbre += 1; // équivalent à nbre = nbre + 1;  
8     }  
9 }
```

```
1 // Affichage des nombres de 0 à 10 avec for  
Run | Debug  
2 void main() {  
3     for (int nbre = 0; nbre < 11; nbre++) {  
4         print(nbre);  
5     }  
6 }
```

Quand on utilise l'instruction **break** dans une boucle celle-ci est stoppée. Quant à l'instruction **continue** elle repart à la valeur suivante de la boucle. Pour plus d'informations voir [boucles](#).

Exercice: Ecrire un programme qui affiche la liste des nombres pairs entre un nombre de départ et de fin. Le programme doit se terminer dès que lors du parcours il rencontre son nombre interdit qui est 100.

Les fonctions et procédures

Pour créer des fonctions et procédures en Dart, il faut utiliser la syntaxe ci-dessous:

Syntaxe des fonctions

```
typeDeRetour nomDeLaFonction(paramètres,...) {  
  
    code d'exécution de la fonction  
  
    return valeurDeRetour;  
}
```



```
1  int carre(int nbre) {  
2      return nbre * nbre;  
3  }  
4  
5  void main() {  
6      final int n = 2;  
7      print("Le carré de $n est ${carre(n)}"); // Affichera Le carré de 2 est 4  
8  }
```

Syntaxe des procédures

```
void nomDeLaProcédure(paramètres,...) {  
  
    code d'exécution de la procédure  
}
```



```
1  void hello(String nom) {  
2      print("Hello $nom");  
3  }  
4  
5  void main() {  
6      hello("Toto"); // Affichera 'Hello Toto'  
7  }  
8
```

Les énumérations

Les énumérations sont définies avec le mot clé ***enum*** et doivent respecter la syntaxe suivante:

Syntaxe de enum

```
enum Nom {  
    valeur constante 1  
    valeur constante 2  
    ...  
}
```

```
1 enum Category { majeur, mineur }  
2  
3 final age = 20;  
4  
5 void main() {  
6     Category category = age >= 18 ? Category.majeur : Category.mineur;  
7  
8     if (category == Category.majeur) {  
9         print("Vous êtes majeur.");  
10    } else {  
11        print("Vous êtes mineur.");  
12    }  
13 }
```




Vous êtes majeur.

Exercice: Ecrire un programme qui affiche la tranche d'âge en fonction de la catégorie qui est une valeur d'enum de nom Category avec comme valeurs poussin (de 6 à 7 ans), pupille (de 8 à 9 ans), minime (de 10 à 11 ans) et cadet (après 12 ans).


Les importations

Dans un projet, il arrivera qu'on ait besoin d'importer des modules internes de Dart, des packages ou des fonctions ou classes de d'autres fichiers. On utilisera les syntaxes suivantes:



```
1 // Pour importer les bibliothèques internes à Dart, il faut précéder de 'dart:' le nom de la bibliothèque à importer.
2 import "dart:io";
3
4 // Pour importer des bibliothèques externes installées, il faut précéder de 'package:' le nom du fichier de base du package(module).
5 import 'package:flutter/material.dart';
6
7 // Pour importer des fichiers de code, il faut préciser le chemin d'accès au fichier dans import.
8 import "package:tp1_premier_programme_dart/test.dart";
9
```

On peut importer un module en lui assignant un préfixe ou importer toute une partie de ses fonctions.



```
1 // Utiliser le mot clé 'as' pour définir un préfixe à spécifier lors de l'utilisation d'une bibliothèque
2 import "dart:io" as dartIo; // Toutes les fonctions de dart:io devront être précédées de dartIo
3
4 // Utiliser le mot clé 'show' pour importer une partie des fonctions ou classes d'un module
5 import "dart:io" show File, Directory; // Seules les classes File et Directory de dart:io seront disponibles les autres non
6
7 // Utiliser le mot clé 'hide' pour importer tout sauf une partie des fonctions ou classes d'un module
8 import "dart:io" hide File, Directory; // Toutes les classes de "dart:io" seront disponibles sauf File et Directory
```

L'asynchrone

Pour créer des procédures ou fonctions asynchrones, on utilisera les mots clés *async/await* avec la syntaxe:

Syntaxe des procédures asynchrones

```
Future<void> nomDeLaProcédure(paramètres,...) async {  
    code d'exécution de la procédure avec await pour les  
    attentes de fin d'exécution  
}
```

```
1 Future<void> direHelloApresUneSeconde() async {  
2     // Attente de la fin de la Future qui est asynchrone  
3     await Future.delayed(Duration(seconds: 1));  
4     print("Hello");  
5 }  
6  
7 void main() {  
8     direHelloApresUneSeconde();  
9     print("Je suis dans le main.");  
10 }
```



Je suis dans le main.
Hello

Les méthodes asynchrones sont de type *Future* c'est pourquoi le retour n'est pas *void* mais *Future < void >*.

Syntaxe des fonctions asynchrones

```
Future<typeDeRetour> nomDeLaFonction(paramètres,...) async {  
    code d'exécution de la fonction avec await pour les  
    attentes de fin d'exécution  
    return valeurDeRetour;  
}
```

```
1 Future<String> renvoyerEnLettreMajusculeApresUneSeconde(String message) async {  
2     // Attente de la fin de la Future qui est asynchrone  
3     await Future.delayed(Duration(seconds: 1));  
4     return message.toUpperCase();  
5 }  
6  
7 void main() {  
8     renvoyerEnLettreMajusculeApresUneSeconde("Toto")  
9         .then((response) => {print(response)});  
10    print("Je suis dans le main.");  
11 }
```



Je suis dans le main.
TOTO

Une fonction asynchrone renvoie un type *Future* < *typeDeRetour* >. On peut utiliser *then/catch* pour récupérer la réponse de la fonction asynchrone et effectuer les traitements sur celle-ci.

Les exceptions

Pour générer des erreurs indiquant que quelque chose d'inattendu s'est produit, on lèvera une exception.

Pour capturer une exception et éviter que le programme ne cesse de fonctionner on utilisera les mots clés *try*, *on* et *catch* avec les syntaxes suivantes:

Syntaxe des exceptions

```
throw TypeDException(message);
```

Ou on peut utiliser

```
throw "message d'erreur";
```

Syntaxe capture des exceptions

```
try {  
    code à exécuter si aucune exception  
} on TypeException {  
    Code a exécuter si une exception spécifique se produit  
} on Exception catch(e) {  
    A utiliser si on veut traiter tout autre cas d'exception  
} catch (e) {  
    Pour capturer tout autre erreur  
}
```

Les exceptions

```
1 double division(int nbre1, int nbre2) {  
2   // On lève une exception car la division par 0 est impossible  
3   if (nbre2 == 0) {  
4     // On aurait pu utiliser throw "nbre2 doit être non nul.";  
5     throw UnsupportedError("nbre2 doit être non nul.");  
6   }  
7   return nbre1 / nbre2;  
8 }  
9  
10 void main() {  
11   print(division(2, 3));  
12   print(division(2, 0));  
13 }
```

```
0.6666666666666666  
DartPad caught unhandled String:  
nbre2 doit être non nul.  
https://storage.googleapis.com/nbd_artifacts/3.5.4/dart_sdk.js 12007:11  
Object.throw_  
blob:null/b35987c3-2920-4df2-9811-8203101e8063 375:17  
Object.division  
blob:null/b35987c3-2920-4df2-9811-8203101e8063 381:21      main$0  
blob:null/b35987c3-2920-4df2-9811-8203101e8063 414:18      <fn>  
https://storage.googleapis.com/nbd_artifacts/3.5.4/dart_sdk.js 48513:14  
_rootRun  
https://storage.googleapis.com/nbd_artifacts/3.5.4/dart_sdk.js 47560:14
```

Exception Non
Capturée

```
1 double division(int nbre1, int nbre2) {  
2   // On lève une exception car la division par 0 est impossible  
3   if (nbre2 == 0) {  
4     // On aurait pu utiliser throw "nbre2 doit être non nul.";  
5     throw UnsupportedError("nbre2 doit être non nul.");  
6   }  
7   return nbre1 / nbre2;  
8 }  
9  
10 void main() {  
11   try {  
12     int nbre1 = 2;  
13     int nbre2 = 0;  
14     double resultat = division(nbre1, nbre2);  
15     print("La division de $nbre1 par $nbre2 est: $resultat");  
16   } on UnsupportedError {  
17     print("La division par zéro est impossible.");  
18   } catch (err) {  
19     print("Une erreur a été rencontrée $err");  
20   }  
21 }
```

La division par zéro est impossible.

Exception Capturée

La programmation orientée objet

La programmation orientée objet en Dart a une syntaxe assez similaire à celle des autres langages.

Syntaxe des classes

```
class NomDeLaClasse {  
    définition des attributs de la classe  
    les variables privées sont précédées de _  
  
    définition du constructeur de la classe  
  
    définition des méthodes et fonctions  
}
```

```
John  
Doé  
Je suis John Doé mon secret est john  
Je suis Toto Oro mon secret est toto
```

```
1 class Personne {  
2     late String nom;  
3     late String prenom;  
4     // Garder le secret privé  
5     late String _secret;  
6  
7     // Une méthode plus rapide est d'utiliser le code ci-dessous  
8     // Personne({required this.nom, required this.prenom, required this.secret})  
9     Personne(String nom, String prenom, String secret) {  
10         this.nom = nom;  
11         this.prenom = prenom;  
12         this._secret = secret;  
13     }  
14  
15     String get secret {  
16         return this._secret;  
17     }  
18  
19     void description() {  
20         print("Je suis $nom $prenom mon secret est $_secret");  
21     }  
22 }  
23  
24 void main() {  
25     // Création d'une personne  
26     Personne personne1 = new Personne("John", "Doé", "john");  
27     print(personne1.nom);  
28     print(personne1.prenom);  
29     personne1.description();  
30     // Création d'une autre personne  
31     Personne personne2 = new Personne("Toto", "Oro", "toto");  
32     personne2.description();  
33 }  
34
```

La programmation orientée objet

Pour faire de l'héritage on utilisera le mots clé *extends*. Pour plus d'informations sur les classes visiter [Classes en Dart](#).

```
1 // Définition d'une classe Elève qui hérite de la classe Personne
2 // La classe mère Personne sera accessible avec le mot clé super.
3 class Etudiant extends Personne {
4     // Ajout des attributs supplémentaire à l'étudiant
5     late String niveau;
6     late int age;
7
8     // Constructeur avec initialisation de l'élève comme étant une personne
9     Etudiant(String nom, String prenom, String secret, String niveau, int age)
10         : super(nom: nom, prenom: prenom, secret: secret) {
11         this.niveau = niveau;
12         this.age = age;
13     }
14     // Code de constructeur équivalent
15     // Eleve(
16     //     {required super.nom,
17     //     required super.prenom,
18     //     required super.secret,
19     //     required String this.classe,
20     //     required int this.age})
21     // : super();
22
23     // Utilisation de @override pour changer une méthode existante du parent
24     @override
25     void description() {
26         // Exécution de la fonction description parente
27         super.description();
28         // Affichage du niveau propre à la classe enfant
29         print("Je étudiant avec un niveau d'études $niveau");
30     }
31
32     // Définition d'une fonction propre aux étudiants
33     void afficherMonAge() {
34         print("J'ai $age ans.");
35     }
36 }
```

```
1 void main() {
2     // Création d'une personne
3     Personne personne = new Personne(nom: "John", prenom: "Doé", secret: "john");
4     personne.description();
5     // Création d'un étudiant
6     Etudiant etudiant = Etudiant("Toto", "Test", "toto", "ingénieur", 18);
7     etudiant.description();
8     etudiant.afficherMonAge();
9 }
10
```

Personne John Doé avec pour secret john créée !
Je suis John Doé mon secret est john
Personne Toto Test avec pour secret toto créée !
Je suis Toto Test mon secret est toto
Je étudiant avec un niveau d'études ingénieur
J'ai 18 ans.

FIN DU CHAPITRE 2