



Le langage Dart

Le site officiel de Dart

Le site officiel de Dart (<https://dart.dev>) est une ressource essentielle pour les développeurs qui souhaitent apprendre, utiliser et contribuer au langage de programmation Dart.

Il offre une multitude d'informations, d'outils et de ressources pour les utilisateurs de tous niveaux.

- **Documentation complète**
- **Exemples de code**
- **Outils de développement**
- **Communauté active**
- **Téléchargements**
- **Blog et actualités**



Le DartPad

DartPad est un éditeur de code en ligne gratuit conçu spécifiquement pour écrire et exécuter du code Dart. <https://dartpad.dev>

Développé par Google, il offre une interface simple et intuitive qui permet aux développeurs de :

- Écrire du code Dart
- Tester et exécuter le code
- Partager le code
- Accéder à des exemples de code
- Intégration avec d'autres outils

Il est d'ailleurs l'outil que nous allons utiliser pour apprendre Dart





Les bases de Dart

Le void main()

La fonction void main() est le point de départ de toute application Dart autonome. Elle marque le début de l'exécution du programme et est utilisée pour organiser et exécuter les instructions du programme.

Elle est composée de:

- void qui indique qu'il n'y a pas de valeur de retour
- main, le nom de la fonction(). Toutes les instructions du programme se feront à l'intérieur ou seront appelées par cette fonction.
- Les parenthèses vides indiquent que nous ne passons pas de paramètres ou arguments dans cette fonction.
- Pour chaque fonction, mais nous y reviendrons, le code s'effectue entre les accolades.

```
void main() {  
    // Instructions du programme  
}
```



Le point virgule

En Dart, comme dans de nombreux autres langages de programmation, le point-virgule (;) est utilisé pour terminer une instruction.

C'est un caractère essentiel pour indiquer la fin d'une ligne de code, permettant ainsi au compilateur de comprendre où se termine une instruction et où commence la suivante.

En enlevant un point virgule dans cette déclaration, nous avons une erreur.

```
void main() {  
    print('Salut les codeurs!');  
}
```



Le Commentaire

Les commentaires en Dart sont des annotations textuelles qui permettent d'ajouter des explications et des informations à votre code, sans affecter son exécution.

Ils ne seront donc pas lus durant la compilation du code

Ils sont essentiels pour améliorer la lisibilité, la compréhension et la maintenabilité de votre code, en particulier pour les projets complexes ou collaboratifs.

Deux types de commentaires existent en Dart :

1. Commentaires sur une seule ligne :
2. Commentaires multilignes :



Le Commentaire

Commentaires sur une seule ligne :

- Ils commencent par le symbole `//`.
- Ils s'étendent jusqu'à la fin de la ligne.
- Ils sont utilisés pour commenter de courts fragments de code ou des notes brèves.

```
// Cette variable stocke le nom d'un  
utilisateur.
```

```
String nomUtilisateur = "Alice";
```



Le Commentaire

Commentaires multilignes :

- Ils commencent par le symbole `/*` et se terminent par `*/`.
- Ils peuvent s'étendre sur plusieurs lignes.
- Ils sont utilisés pour commenter des blocs de code plus longs ou des descriptions détaillées.

```
/*  
Ce programme Dart affiche un message de  
salutation.  
Il utilise une fonction principale 'main' pour  
exécuter le code.  
Le message est affiché en utilisant la fonction  
'print'.  
*/  
void main() {  
    print('Bonjour, les codeurs!');  
}
```



Le Commentaire

Les bonnes pratiques pour les commentaires en Dart :

- Soyez précis et concis : Les commentaires doivent expliquer clairement le code qu'ils décrivent, sans être trop pompeux.
- Utilisez des mots-clés pertinents : Choisissez des mots-clés qui reflètent le but du code commenté.
- Commentez la logique, pas le code évident : Ne commentez pas le code qui est facile à comprendre, concentrez-vous sur les parties complexes ou non évidentes.
- Maintenez les commentaires à jour : Lorsque vous modifiez le code, assurez-vous de mettre à jour les commentaires correspondants.



Le print

La fonction `print` est utilisée pour afficher du texte ou des valeurs à la console.

C'est une fonction très utile pour le débogage et pour donner des informations en cours d'exécution du programme.

Syntaxe: `print(expression)`

L'argument `expression` peut être une chaîne de caractères, une variable, ou toute autre valeur qui peut être convertie en chaîne de caractères.

```
void main() {  
    print('Salut les codeurs!');  
}
```



Les variables

Une variable en programmation est un espace mémoire nommé utilisé pour stocker une valeur.

Elle permet d'associer un nom à une donnée et de la manipuler dans le code.

Cette donnée est identifiée par le nom de la variable et peut être ensuite utilisée et modifiée.

Imaginez cela comme une boîte avec un nom contenant un objet. Vous pouvez dans cette boîte:

- La créer (déclarer)
- La retrouver via son nom
- Modifier son contenu



Les types de variable

Dart est un langage à typage statique, ce qui signifie que chaque variable doit avoir un type associé. Le type d'une variable détermine le type de données qu'elle peut stocker. Voici les principaux types de variables en Dart :

- `int` : Nombres entiers (ex: 1, 100, -52)
- `double` : Nombres décimaux (ex: 3.14, 9.75, -2.5)
- `String` : Chaînes de caractères (ex: "Bonjour", "Ceci est une phrase", "123MainStreet")
- `bool` : Valeurs booléennes (Vrai ou Faux)
- `List` : Listes ordonnées d'éléments (ex: [1, 2, 3], ["Pomme", "Orange", "Banane"], [true, false, true])
- `Set` : Ensemble d'éléments non ordonnés et uniques (ex: {1, 2, 3}, {"Pomme", "Orange", "Banane"}, {true, false})
- `Map` : Dictionnaire de clés-valeurs (ex: {"nom": "Pierre", "age": 30, "ville": "Paris"}, {"Pomme": 1.5, "Orange": 2.2, "Banane": 1.8}, {"estVrai": true, "estFaux": false})



Déclarer une variable

Pour déclarer une variable en Dart, on utilise:

- le mot-clé var ou le type de donnée
- nom de la variable
- assignation avec le signe d'égalité
- la valeur initiale (facultatif) :

```
var nomVariable = valeurInitiale;
```

```
String maPremiereVariable = "Salut les  
codeurs";
```



Accéder à une variable

Une fois déclarée, vous pouvez utiliser cette variable en appelant son nom:

```
String maPremiereVariable = "Salut les  
codeurs";  
  
print(maPremiereVariable);
```



Modifier une variable

Vous pouvez aussi modifier une variable:

- entrez le nom de la variable
- suivi du signe =
- Entrez la nouvelle valeur en faisant attention à ce que le type soit le même
- Attention à ne pas mettre le type ou var en début de déclaration

```
String maPremiereVariable = "Salut les  
codeurs";
```

```
print(maPremiereVariable);
```

```
maPremiereVariable = "Salut";
```

```
print(maPremiereVariable);
```



Les conventions de nommage

Règles de base du nommage des variables :

- Les noms de variables peuvent commencer par des lettres majuscules (pour les classes) ou minuscules (pour les variables et fonctions) ou encore (_) pour les variables privées.
- Les caractères spéciaux autorisés sont les lettres de l'alphabet (a-z, A-Z), les chiffres (0-9), le trait de soulignement (_), et le dollar (\$, uniquement en fin de nom pour conventions de bibliothèques).
- Les noms de variables ne peuvent pas être des mots-clés réservés en Dart (comme var, if, for, else, etc.).



Les conventions de nommage

Conventions de nommage recommandées :

- Utilisez des noms descriptifs et significatifs. Le nom de la variable doit refléter son contenu ou sa fonction dans le code.
- Favorisez la clarté sur la concision. Un nom plus long mais explicite est préférable à un nom court et obscur.
- Utilisez le camelCase pour les noms composés de plusieurs mots. Mettez en majuscule la première lettre de chaque mot sauf le premier (ex: totalCommentaire, nombreDeClic).
- Utilisez le snake_case pour les noms avec plusieurs mots séparés par des traits de soulignement (moins courant en Dart). (ex: total_commentaire, nombre_de_clic).
- Utilisez des préfixes ou des suffixes pour les variables de type similaire. Par exemple, vous pouvez utiliser un préfixe est pour les variables booléennes (estValide, estConnecte).



Les conventions de nommage

Exemples de bons noms de variables :

- prenom
- nombreDeChapitres
- estEnregistre
- calculerSurface
- prixTotal
- listeProduits
- utilisateurCourant
- tempsRestantSeconde

Exemples de mauvais noms de variables :

- x (trop vague sauf pour les boucles)
- temp (peu descriptif)
- 1nombre (ne peut pas commencer par un chiffre)
- total-prix (mots reliés par un trait d'union)



Variable constante (const)

Définition: Les variables const sont immuables, ce qui signifie que leur valeur ne peut pas être modifiée après leur initialisation.

Syntaxe: `const type nomVariable = valeurInitiale;`

Utilisation:

- Utilisées pour stocker des valeurs qui ne changent jamais pendant l'exécution du programme (ex: constantes mathématiques, identificateurs uniques).
- Améliorent la lisibilité et la maintenabilité du code en indiquant clairement que la valeur est fixe.
- Permettent l'optimisation du compilateur car la valeur est connue à la compilation.

```
const PI = 3.14159;  
// Valeur constante de Pi  
  
const NOM_APPLICATION =  
"MonApplication";  
// Nom constant de l'application
```



Variable finale (final)

Définition: Les variables final sont également immuables, mais leur valeur peut être modifiée pendant l'initialisation de la variable.

Syntaxe: final type nomVariable = valeurInitiale;

Utilisation:

- Similaires aux variables const, mais permettent une initialisation flexible pendant la déclaration.
- Utilisées pour stocker des valeurs qui sont initialisées une seule fois au début du programme et qui ne changent plus ensuite.

```
final String nom = "Alice";  
  
final int nombreHeuresTravillees = 40;
```



Variable privée (private)

Définition: Les variables private sont uniquement accessibles dans le bloc de code où elles sont déclarées.

Syntaxe: private type _nomVariable = valeurInitiale;

Utilisation:

- Utilisées pour encapsuler des données internes d'une classe ou d'une fonction et les protéger contre un accès non intentionnel depuis l'extérieur.
- Améliorent la modularité et la sécurité du code en limitant l'accès aux données sensibles.

```
class Personne {  
    private String _nom;  
    private int _age;  
  
    public String getNom() {  
        return _nom;  
    }  
    public void setNom(String nouveauNom) {  
        _nom = nouveauNom;  
    }  
}
```





Les types de variable

String

En Dart, les String sont des types de données utilisés pour représenter du texte. Les chaînes de caractères sont des éléments fondamentaux de la programmation, utilisées pour stocker et manipuler des données textuelles telles que des messages, des noms, des adresses, etc.

Déclaration de Chaînes de Caractères

Les chaînes de caractères peuvent être déclarées en utilisant soit des guillemets simples (' '), soit des guillemets doubles (" ").

```
String maPremiereVariable = "Salut les  
codeurs";
```



String sur plusieurs lignes

En utilisant les triples guillemets ("""), vous pouvez écrire une String sur plusieurs lignes:

```
String multilineString = ""  
Je suis une String  
Et sur plusieurs lignes !!  
""";  
print(multilineString);
```



String: Les caractères d'échappement

En Dart, le caractère \ (antislash) est utilisé pour plusieurs fonctions importantes :

Le \ permet d'indiquer au compilateur Dart qu'un caractère suivant doit être interprété littéralement, même s'il a une signification spéciale dans les chaînes de caractères.

- \' pour une apostrophe (guillemet simple)
- \" pour un guillemet double
- \\ pour un antislash
- \n pour un saut de ligne
- \t pour une tabulation

```
String message = "Il a dit : \"Bonjour\".";
```



String: Concaténation

La concaténation de chaînes de caractères en Dart peut être réalisée en utilisant l'opérateur +

```
String firstName = 'Alice';  
String lastName = 'Smith';  
  
String fullName = firstName + ' ' + lastName;  
  
print(fullName);
```



String: Interpolation

l'interpolation de chaînes de caractères se fait quand à elle avec le symbole \$ qui précède la variable au sein d'une autre String.

```
String firstName = 'Alice';  
String lastName = 'Smith';  
  
String fullName = '$firstName $lastName';  
  
print(fullName);
```



String: longueur

Vous pouvez obtenir la longueur d'une chaîne de caractères en utilisant la propriété `length`.

```
String firstName = 'Matthieu';  
print(firstName.length);
```



String: Accès aux caractères

Dans chaîne de caractère, il y a chaîne. Imaginez donc votre String comme un ensemble de maillons mis bout à bout.

Et si on voulait accéder à un des maillons?

Vous pouvez accéder aux caractères individuels d'une chaîne de caractères en utilisant entre crochets l'index de caractère, qui commence à partir de zéro.

```
String firstName = 'Matthieu';  
print(firstName[0]); //M  
print(firstName[3]); //t
```



String: Extraire une sous chaine

Syntaxe:

chaine.substring(startIndex, endIndex)

```
String helloString = 'Salut les codeurs';  
  
var substring = helloString.substring(1, 7);  
  
print(substring);
```



String: Remplacer du texte

Syntaxe:

chaine.replaceAll(oldText, newText)

```
String helloString = 'Salut les codeurs';  
  
var replacement = helloString.replaceAll("e",  
    "* Ici un e *");  
print(replacement);
```



String: Majuscule ou Minuscule

Syntaxe:

chaine.toUpperCase()
chaine.toLowerCase()

```
String helloString = 'SaLuT lEs COdEurs';  
  
print(helloString.toUpperCase());  
print(helloString.toLowerCase());
```



String: Autres méthodes utiles

- **Vérifier si vide:** chaine.isEmpty
- **Commence par:** chaine.startsWith(expression)
- **Finit par:** chaine.endsWith(expression)
- **Contient:** chaine.contains(expression)
- **Couper:** chaine.trim(espace de début)

```
String helloString = 'Salut les codeurs';
```

```
print(helloString.startsWith("S"));  
print(helloString.endsWith("X"));  
print(helloString.contains("LES"));
```

```
String toBeTrimmed = "  Et oui.  ";  
print(toBeTrimmed);  
print(toBeTrimmed.trim());
```



Int: Les nombres entiers

En Dart, les int sont des types de données entiers utilisés pour représenter des nombres entiers sans partie décimale. Les nombres entiers sont utilisés dans de nombreuses applications pour compter, indexer, effectuer des calculs mathématiques et plus encore. Dans cette explication, je vais détailler les fonctionnalités et les opérations courantes sur les nombres entiers en Dart.

```
int age = 30;  
  
int count = 100;
```



Int: Conversion

Vous pouvez convertir un nombre entier en une chaîne de caractères en utilisant la méthode `toString()`.

Ou encore

Vous pouvez également convertir une chaîne de caractères représentant un nombre entier en un **int** en utilisant la fonction **`int.parse()`**.

```
int age = 30;

String ageString = age.toString();

print(ageString);

int ageBackInt = int.parse(ageString);

print(ageBackInt);
```



double: Les nombres décimaux

En Dart, le type `double` est utilisé pour représenter des nombres à virgule flottante, c'est-à-dire des nombres qui ont une partie fractionnaire.

Les `double` sont couramment utilisés pour les calculs qui nécessitent une précision décimale, comme les opérations financières, les mesures scientifiques, etc.

```
double pi = 3.14;  
double temperature = -5.6;  
double height = 1.75;
```



double: méthodes utiles

Les double en Dart disposent de plusieurs méthodes et propriétés utiles :

- `toString()`: Convertit un double en une chaîne de caractères.
- `toInt()`: Convertit un double en un entier en supprimant la partie fractionnaire (arrondi vers zéro).
- `abs()`: Renvoie la valeur absolue du double.
- `ceil()`: Renvoie le plus petit entier supérieur ou égal au double.
- `floor()`: Renvoie le plus grand entier inférieur ou égal au double.
- `round()`: Renvoie l'entier le plus proche du double.

```
double pi = 3.14;
```

```
// A vous d'essayer avec un print pour voir  
les résultats
```



double: précision

Les double sont des nombres à virgule flottante en précision double, ce qui signifie qu'ils peuvent représenter des nombres très grands ou très petits, mais avec une précision limitée. Cela peut entraîner des erreurs d'arrondi dans certains calculs.

```
double result = 0.1 + 0.2;  
print(result);  
  
// Affiche 0.30000000000000004 au lieu de 0.3
```



bool: vrai ou faux

Les booléens (ou bool en Dart) sont des types de données utilisés pour représenter des valeurs logiques : true (vrai) et false (faux).

Ils sont essentiels pour les opérations de comparaison, les contrôles de flux (comme les structures conditionnelles et les boucles), et la logique de programmation en général.

```
bool isRaining = true;
```

```
bool isSunny = false;
```



bool: négation du bool

Il s'agit ici d'un opérateur logique, nous y reviendrons dans le chapitre sur les opérateurs:

En plaçant un ! avant son nom de variable, vous inversez la valeur booléenne

```
bool isRaining = true;  
  
bool isNotRaining = !isRaining;  
// false
```



Le dynamic

Le type dynamic en Dart est un type spécial qui permet de stocker des valeurs de n'importe quel type.

Lorsque vous utilisez dynamic, le type de la variable est déterminé au moment de l'exécution plutôt qu'à la compilation.

Cela permet une grande flexibilité, mais peut également introduire des risques de bugs si le type réel des valeurs n'est pas géré correctement.

```
dynamic value = 'Bonjour';  
print(value); // Affiche: Bonjour
```

```
value = 123;  
print(value); // Affiche: 123
```

```
value = true;  
print(value); // Affiche: true
```



Le dynamic: risques

L'utilisation de dynamic offre une grande flexibilité, mais elle comporte des risques :

1. **Erreurs d'Exécution** : Étant donné que les vérifications de type sont reportées à l'exécution, vous risquez d'obtenir des erreurs si vous essayez d'appeler une méthode qui n'existe pas pour le type réel de la variable.
2. **Perte de Lisibilité et de Maintenabilité** : Le code devient plus difficile à lire et à maintenir, car le type des variables n'est pas clair.

```
dynamic value = 'Bonjour';  
  
dynamic value = 'Bonjour';  
  
value = 123;  
  
print(value.toUpperCase()); // Erreur
```



Le dynamic: Bonne pratiques

Pour éviter les problèmes liés à l'utilisation de dynamic, voici quelques bonnes pratiques :

1. **Limiter l'Utilisation de dynamic** : Utilisez dynamic seulement lorsque cela est nécessaire. Préférez les types explicites pour la plupart des cas.
2. **Vérification de Type** : Utilisez les opérateurs `is` et `as` pour vérifier et convertir les types lors de l'utilisation de dynamic.
3. **Commentaires et Documentation** : Ajoutez des commentaires et documentez votre code pour indiquer pourquoi vous utilisez dynamic et quel type de données est attendu.

```
dynamic value = 'Bonjour';

if (value is String) {
    print(value.toUpperCase());
} else {
    print('value n'est pas une chaîne de caractères');
}
```



Le null

Le concept de null en Dart représente l'absence de valeur.

Il est utilisé pour indiquer qu'une variable n'a pas été initialisée avec une valeur valide.

Comprendre et gérer correctement le null est essentiel pour éviter les erreurs courantes dans la programmation, telles que les exceptions de type "null pointer". Ce qui pourrait causer des crash d'application



Le nullable

En Dart, vous pouvez déclarer une variable sans lui assigner immédiatement une valeur, ce qui la rendra null par défaut si elle n'est pas explicitement initialisée.

Le point d'interrogation (?) après le type indique que la variable peut être **null**. Vous la rendez nullable.

En Dart, les variables non nullable doivent être initialisées avec une valeur valide.

```
int age = 30; // Variable non nullable, doit être initialisée

int? nullableAge; // Variable nullable, peut être null
```



Le null safety

Depuis la version 2.12 de Dart, le langage inclut un système de null safety qui aide à prévenir les erreurs liées aux valeurs null.

Avec null safety, les types par défaut ne peuvent pas être null à moins qu'ils ne soient explicitement marqués comme pouvant l'être en ajoutant ? après le type.

```
int? nullableNumber;  
nullableNumber = 5;  
// nullableNumber peut être un entier ou null  
nullableNumber = null;  
// C'est valide car nullableNumber est nullable  
  
int nonNullableNumber = 10;  
nonNullableNumber = null;  
// Erreur de compilation, nonNullableNumber ne  
// peut pas être null
```



Le null: accès sécurisé

L'opérateur d'accès sécurisé (?.) permet d'appeler des méthodes ou d'accéder à des propriétés d'un objet nullable uniquement si cet objet n'est pas null.

```
String? message;  
print(message?.length);  
// Affiche: null  
  
message = "Bonjour";  
print(message?.length);  
// Affiche: 7
```



Le null assertion

L'opérateur d'assertion null (!) permet de forcer une expression nullable à être traitée comme non nullable, ce qui peut entraîner une erreur à l'exécution si la valeur est effectivement null

```
String? message;  
message = "Bonjour";  
print(message!);  
// Affiche: Bonjour  
  
message = null;  
print(message!);  
// Lève une erreur à l'exécution
```



Le late: intiation tardive

L'initialisation tardive (late) permet de déclarer une variable non nullable qui sera initialisée plus tard, avant sa première utilisation.

Cela peut être utile pour éviter les problèmes de null safety lors de l'initialisation différée

```
late String name;  
name = "Alice";
```

```
print(name);  
// Affiche: Alice
```

```
// Si vous accédez à `name` avant de  
l'initialiser, une erreur sera levée
```



Les Listes

En Dart, une List est une collection ordonnée d'éléments.

Les listes sont des structures de données courantes qui permettent de stocker des ensembles de valeurs, comme des numéros, des chaînes de caractères ou des objets.

Les éléments dans une liste peuvent être accédés par leur index, ce qui rend les opérations de lecture et de modification très efficaces.



Les Listes: Déclaration

Vous pouvez déclarer et initialiser une liste en utilisant les crochets [].

Une liste peut être typée ou non typée.

Par défaut, une liste peut contenir des éléments de n'importe quel type, mais il est recommandé d'utiliser des listes typées pour plus de sécurité et de clarté.

```
List myList = [1, 'two', 3.0, true]; print(myList);  
// Affiche: [1, two, 3.0, true]
```

```
List<int> intList = [1, 2, 3, 4, 5];  
print(intList);  
// Affiche: [1, 2, 3, 4, 5]
```

```
List<String> stringList = ['Alice', 'Bob',  
'Charlie'];  
print(stringList);  
// Affiche: [Alice, Bob, Charlie]
```



Les Listes: Accès

Les éléments d'une liste peuvent être accédés par leur index, **qui commence à 0**.

```
List<String> stringList = ['Alice', 'Bob',  
    'Charlie'];  
  
print(stringList[0]);  
    // Affiche: Alice  
  
print(stringList[2]);  
    // Affiche: Charlie
```



Les Listes: Compter

La longueur de la liste peut être obtenue en utilisant la propriété `length`.

```
List<int> intList = [1, 2, 3, 4, 5];  
  
print(intList.length);  
// Affiche: 5
```



Les Listes: Modification

Vous pouvez modifier les éléments d'une liste en accédant à l'index correspondant et en assignant une nouvelle valeur.

```
List<int> intList = [1, 2, 3];  
  
intList[1] = 42;  
  
print(intList);  
// Affiche: [1, 42, 3]
```



Les Listes: Ajout

Vous pouvez ajouter des éléments à une liste en utilisant les méthodes `add` et `addAll`.

```
List<int> intList = [1, 2, 3];  
intList.add(4);  
print(intList);  
// Affiche: [1, 2, 3, 4]  
  
intList.addAll([5, 6]);  
print(intList);  
// Affiche: [1, 2, 3, 4, 5, 6]
```



Les Listes: Insertion

Pour insérer un élément à un index spécifique, utilisez la méthode `insert(index, valeur)`.

```
List<int> intList = [1, 2, 3];  
  
intList.insert(1, 42);  
  
print(intList);  
// Affiche: [1, 42, 2, 3]
```



Les Listes: Suppression

Vous pouvez supprimer des éléments en utilisant les méthodes:

- `remove(value)`
- `removeAt(index)`
- `removeWhere(condition)`

```
List<int> intList = [1, 2, 3, 4];
intList.remove(2);
print(intList);
// Affiche: [1, 3, 4]

intList.removeAt(1);
print(intList);
// Affiche: [1, 4]

intList.removeWhere((item) => item > 3);
print(intList);
// Affiche: [1]
```



Les Listes: non modifiable

Si vous voulez créer une liste dont le contenu ne peut pas être modifié, vous pouvez utiliser la méthode `List.unmodifiable`.

```
var immutableList = List.unmodifiable([1, 2, 3]);  
  
print(immutableList);  
  
immutableList[0] = 4;  
// Lève une erreur
```



Les Listes: autres méthodes utiles

Dart fournit plusieurs fonctions utiles pour travailler avec les listes, telles que :

- `length`: Renvoie la longueur de la liste
- `isEmpty`: Vérifie si la liste est vide
- `first`: Renvoie le premier élément de la liste
- `last`: Renvoie le dernier élément de la liste
- `contains`: Vérifie si la liste contient un élément donné
- `indexOf`: Renvoie l'index de la première occurrence d'un élément donné dans la liste
- `lastIndexOf`: Renvoie l'index de la dernière occurrence d'un élément donné dans la liste
- `sort()`: Trie les éléments de la liste



Les Map

En Dart, un Map est une collection d'éléments associatifs où chaque élément est composé d'une paire clé-valeur.

Les clés sont uniques dans un Map, ce qui signifie qu'une clé spécifique ne peut être associée qu'à une seule valeur à la fois.

Les Map sont utiles pour stocker des données structurées où chaque élément est identifiable par une clé unique.



Les Map: Déclaration

Vous pouvez déclarer et initialiser un Map en utilisant les accolades {}.

Vous spécifiez la clé et sa valeur en utilisant la syntaxe clé: valeur.

```
var myMap = {'name': 'Alice', 'age': 30};  
print(myMap);  
// Affiche: {name: Alice, age: 30}  
  
Map<String, int> ages = {'Alice': 30, 'Bob': 25,  
    'Charlie': 35};  
print(ages);  
// Affiche: {Alice: 30, Bob: 25, Charlie: 35}
```



Les Map: Accès

Vous pouvez accéder aux valeurs d'un Map en utilisant leurs clés.

```
var person = {'name': 'Alice', 'age': 30};  
  
print(person['name']);  
// Affiche: Alice  
  
print(person['age']);  
// Affiche: 30
```



Les Map: Compter

La longueur d'un Map correspond au nombre de paires clé-valeur qu'il contient.

Vous pouvez obtenir la longueur d'un Map en utilisant la propriété `length`.

```
var person = {'name': 'Alice', 'age': 30};  
  
print(person.length);  
// Affiche: 2
```



Les Map: Modification

Vous pouvez modifier les valeurs d'un Map en utilisant leurs clés et en lui assignant une nouvelle valeur.

```
var person = {'name': 'Alice', 'age': 30};  
  
person['age'] = 35;  
  
print(person);  
// Affiche: {name: Alice, age: 35}
```



Les Map: Ajout

Pour ajouter une nouvelle paire clé-valeur à un Map, utilisez la même méthode que pour la modification. Si la clé n'existe pas, une nouvelle paire clé-valeur sera créée.

```
var person = {'name': 'Alice', 'age': 30};  
  
person['city'] = 'New York';  
  
print(person);  
// Affiche: {name: Alice, age: 30, city: New York}
```



Les Map: Suppression

Pour supprimer une paire clé-valeur d'un Map, utilisez la méthode `remove(clé)`.

```
var person = {'name': 'Alice', 'age': 30};  
  
    person.remove('age');  
print(person);  
// Affiche: {name: Alice}
```



Les Map: non modifiable

Si vous souhaitez créer un Map dont le contenu ne peut pas être modifié, vous pouvez utiliser la méthode `Map.unmodifiable`

```
var immutableMap = Map.unmodifiable({'name':  
'Alice', 'age': 30});  
  
immutableMap['age'] = 35;  
// Lève une erreur
```



Les Map: autres méthodes utiles

Dart fournit plusieurs fonctions utiles pour travailler avec les listes, telles que :

- isEmpty: Vérifie si la Map est vide
- keys: Renvoie une liste de toutes les clés de la Map
- values: Renvoie une liste de toutes les valeurs de la Map
- containsKey: Vérifie si une clé existe dans la Map
- containsValue: Vérifie si une valeur existe dans la Map
- putIfAbsent: Ajoute une paire clé-valeur à la Map si la clé n'existe pas déjà



Les Set

En Dart, un Set est une collection d'éléments uniques, ce qui signifie qu'aucun élément ne peut apparaître plus d'une fois dans un Set.

Les Set sont utiles lorsque vous avez besoin de stocker des éléments sans vous soucier de leur ordre ou de leur duplication.



Les Set: Déclaration

Vous pouvez déclarer et initialiser un Set en utilisant les accolades {}. Vous spécifiez les éléments du Set en les séparant par des virgules.

```
var mySet = {'apple', 'banana', 'orange'};
print(mySet);
// Affiche: {apple, banana, orange}

Set<String> fruits = {'apple', 'banana',
'orange'};
print(fruits);
// Affiche: {apple, banana, orange}
```



Les Set: Accès

Vous ne pouvez pas accéder directement aux éléments d'un Set par leur index, car les Set ne sont pas ordonnés.

Vous pouvez utiliser la méthode `contains()` pour vérifier si un élément existe dans un Set.

```
Set<String> fruits = {'apple', 'banana',  
    'orange'};  
  
print(fruits.contains('apple'));
```



Les Set: Compter

Comme le Map et le List, vous pouvez grâce à `.length` compter le nombre d'éléments présents dans un Set.

```
Set<String> fruits = {'apple', 'banana',  
    'orange'};  
print(fruits.length);
```



Les Set: Ajout

Vous pouvez ajouter des éléments à un Set en utilisant la méthode `add`.

```
Set<String> fruits = {'apple', 'banana'};

fruits.add('orange');

print(fruits);
// Affiche: {apple, banana, orange}
```



Les Set: Suppression

Vous pouvez supprimer des éléments d'un Set en utilisant la méthode `remove`.

```
Set<String> fruits = {'apple', 'banana', 'orange'};  
  
fruits.remove("banana");  
  
print(fruits);
```



Les Set: autres méthodes utiles

Dart fournit plusieurs fonctions utiles pour travailler avec les Set, telles que :

- isEmpty: Vérifie si le Set est vide
- union: Crée un nouveau Set qui contient tous les éléments des deux Set d'origine
- intersection: Crée un nouveau Set qui contient tous les éléments communs aux deux Set d'origine
- difference: Crée un nouveau Set qui contient tous les éléments du premier Set qui ne sont pas présents dans le deuxième Set
- toSet(): Convertit une liste en un Set (supprime les doublons)





Les opérateurs

L'opérateur d'assignation

L'opérateur d'assignation est principalement utilisé pour affecter ou réaffecter une valeur à une variable.

Nous l'avons déjà vu, il est représenté par le signe =

```
int age = 10;  
  
String name = "Matthieu";  
  
age = 35;  
  
name = "Gérard";
```



Les opérateurs arithmétiques

Les opérateurs arithmétiques en Dart permettent d'effectuer des opérations mathématiques sur des nombres.

Ils sont similaires aux opérateurs arithmétiques que l'on trouve dans d'autres langages de programmation.

- `+`: addition
- `-`: soustraction
- `*`: multiplication
- `/`: division

```
var sum = 5 + 3; // sum est égal à 8

var difference = 10 - 7; // difference est égal à 3

var product = 4 * 6; // product est égal à 24

var quotient = 15 / 3; // quotient est égal à 5.0
```



Les opérateurs arithmétiques

Vous avez remarqué que le résultat d'une division donne automatiquement un double.

Division Entière (~/) : Utilisé pour effectuer une division entière, en ignorant la partie fractionnaire du résultat

Opérateur de Reste ou modulo (%) : Utilisé pour obtenir le reste d'une division.

```
int full = 15 ~/ 6;  
int modulo = 15 % 6;  
  
print(full);  
print(modulo);
```



Les opérateurs arithmétiques

Incrémentation (++) et Décrémentation (--):

Utilisés pour augmenter ou diminuer une valeur de 1.

```
int number = 5;  
number++;  
print(number);  
  
number--;  
print(number);
```



Les opérateurs d'affectation arithmétique

- += addition
- -= soustraction
- *= multiplication
- /= division

```
double number = 5;  
number += 6;  
print(number);  
  
number -= 2;  
print(number);  
  
number *= 7;  
print(number);  
  
number /= 4;  
print(number);
```



Les opérateurs de comparaison

Les opérateurs de comparaison en Dart sont utilisés pour comparer des valeurs et retourner un booléen indiquant si la condition est vraie ou fausse.

- **Égalité (==)** : Vérifie si deux valeurs sont égales.
- **Différence (!=)** : Vérifie si deux valeurs sont différentes.
- **Inférieur (<)** : Vérifie si une valeur est inférieure à une autre.
- **Inférieur ou Égal (<=)** : Vérifie si une valeur est inférieure ou égale à une autre.
- **Supérieur (>)** : Vérifie si une valeur est supérieure à une autre.
- **Supérieur ou Égal (>=)** : Vérifie si une valeur est supérieure ou égale à une autre.

```
int a = 5;  
int b = 7;  
  
print(a == b);  
print(a != b);  
print(a < b);  
print(a <= b);  
print(a > b);  
print(a >= b);
```



Les opérateurs logiques

- Les opérateurs logiques en Dart sont utilisés pour combiner ou inverser des expressions booléennes afin de construire des conditions complexes. Dart prend en charge plusieurs opérateurs logiques qui permettent de manipuler les valeurs booléennes de manière efficace.
- **Et (&&)** : L'opérateur **&&** retourne true si et seulement si les deux expressions booléennes sont true.
- **Ou (||)** : L'opérateur **||** retourne true si au moins l'une des expressions booléennes est true.
- **Non (!)** : L'opérateur **!** inverse la valeur de l'expression booléenne.

```
var x = true;
var y = false;
print(x && y); // Affiche: false

print(x || y); // Affiche: true

print(!x); // Affiche: false
```



Les opérateurs logiques

Court-Circuitage

Dart utilise le court-circuitage logique, ce qui signifie que dans une expression `&&` ou `||`, l'évaluation s'arrête dès qu'il est possible de déterminer le résultat final.

Par exemple, dans une expression `&&`, si la première expression est fausse, la deuxième expression n'est pas évaluée car le résultat global sera toujours faux.

Ordre d'Évaluation

Les expressions logiques sont évaluées de gauche à droite, et les parenthèses peuvent être utilisées pour spécifier explicitement l'ordre d'évaluation.



L'opérateur ternaire

L'opérateur ternaire, également appelé opérateur conditionnel, est une structure de contrôle de flux très utile en Dart.

Syntaxe

condition ? expressionSiVraie : expressionSiFausse;

Fonctionnement

- Condition : Une expression booléenne qui est évaluée. Si cette condition est vraie (true), l'expression qui suit le ? est exécutée. Sinon, l'expression qui suit le : est exécutée.
- Expression Si Vraie : valeur si la condition est vraie.
- Expression Si Fausse : valeur si la condition est fausse.

```
var age = 20;

var estMajeur = age >= 18 ? true : false;

print(estMajeur);
// Affiche: true
```



L'opérateur null aware

Cet opérateur est très utile pour fournir une valeur de secours ou par défaut lorsqu'une expression évalue à null.

Syntaxe

expression1 ?? expression2;

Fonctionnement

- Expression 1 : L'expression qui est évaluée en premier. Si cette expression n'est pas null, elle est renvoyée comme résultat de l'opération.
- Expression 2 : L'expression qui est renvoyée si l'expression 1 est null.

```
String? nom;  
  
var nomAffiche = nom ?? "Invité";  
  
print(nomAffiche);  
// Affiche: Invité
```





Les boucles

Les boucles

Les boucles en Dart permettent d'exécuter un bloc de code de manière répétée jusqu'à ce qu'une condition soit remplie.

Elles sont utiles pour automatiser des tâches répétitives et parcourir des collections de données.

Types de boucles

Dart propose différents types de boucles pour répondre à différents besoins :

- for range
- for ... in
- while
- do ... while



La boucle for range

Permet d'exécuter un bloc de code un nombre déterminé de fois.

Explication :

- for: mot-clé qui indique le début d'une boucle for.
- int i = 0: déclaration d'une variable i de type int initialisée à 0.
- i < 10: condition qui doit être vraie pour que le corps de la boucle soit exécuté. Dans ce cas, la boucle s'exécute tant que i est inférieur à 10.
- i++: instruction qui incrémente la valeur de i de 1 après chaque exécution du corps de la boucle.
- print(i): instruction qui affiche la valeur de i à chaque tour de boucle.

```
for (int i = 0; i < 10; i++) {  
    print(i);  
}
```



La boucle for ... in

Permet de parcourir les éléments d'une collection (list, Map, etc.).

Explication :

- for: mot-clé qui indique le début d'une boucle for...in.
- String student: déclaration d'une variable nom de type String pour stocker chaque élément de la collection.
- in classroom: collection à parcourir, ici une liste de noms.
- print(student): instruction qui affiche chaque nom de la liste à chaque tour de boucle.

```
List<String> classroom = ["Alice", "Bob",  
"Charlie"];
```

```
for (String student in classroom) {  
    print(student);  
}
```



La boucle while

Permet d'exécuter un bloc de code tant qu'une condition est vraie.

Explication :

- `int i = 0`: déclaration d'une variable `i` de type `int` initialisée à 0.
- `while (i < 10)`: condition qui doit être vraie pour que le corps de la boucle soit exécuté. Dans ce cas, la boucle s'exécute tant que `i` est inférieur à 10.
- `print(i)`: instruction qui affiche la valeur de `i` à chaque tour de boucle.
- `i++`: instruction qui incrémente la valeur de `i` de 1 après chaque exécution du corps de la boucle.

```
int i = 0;

while (i < 10) {
    print(i);
    i++;
}
```



La boucle do ... while

Exécute d'abord le corps de la boucle, puis vérifie la condition.

Explication :

- `int i = 0`: déclaration d'une variable `i` de type `int` initialisée à 0.
- `do`: mot-clé qui indique le début d'une boucle `do...while`.
- `print(i)`: instruction qui affiche la valeur de `i` à chaque tour de boucle.
- `i++`: instruction qui incrémente la valeur de `i` de 1 après chaque exécution du corps de la boucle.
- `while (i < 10)`: condition qui doit être vraie pour que la boucle soit exécutée à nouveau. Dans ce cas, la boucle s'exécute tant que `i` est inférieur à 10.

```
int i = 0;
do {
    print(i);
    i++;
} while (i < 10);
```



Quelle boucle choisir?

Le choix de la boucle à utiliser dépend du contexte et de la tâche à accomplir.

Boucle for ... range:

idéale pour un nombre déterminé d'itérations.

Boucle for...in:

idéale pour parcourir des collections.

Boucle while:

idéale pour exécuter un code tant qu'une condition est vraie, sans connaître le nombre d'itérations à l'avance.

Boucle do...while:

similaire à la boucle while, mais le code sera effectué au moins une fois avant vérification de la condition





Les conditions

Les conditions

Les conditions en Dart permettent de contrôler le flux d'exécution d'un programme en fonction de tests booléens.

Elles sont essentielles pour prendre des décisions et exécuter du code de manière conditionnelle.

Types de conditions

Dart propose plusieurs types de conditions pour répondre à différents besoins :

- if
- if ... else
- if ... else if ... else
- switch
- l'opérateur ternaire (que nous avons déjà vu)



La condition if

Permet d'exécuter un bloc de code si une condition est vraie.

Syntaxe :

```
if (condition) {  
    // Code à exécuter si la condition est vraie  
}
```

```
int nombre = 10;  
  
if (nombre > 5) {  
    print("$nombre est supérieur à 5");  
}
```



La condition if .. else

Permet d'exécuter un bloc de code si une condition est vraie.

Syntaxe :

```
if (condition) {  
    // Code à exécuter si la condition est vraie  
} else {  
    //Code a exécuter si la condition est fausse  
}
```

```
int nombre = 10;  
if (nombre > 5) {  
    print("$nombre est supérieur à 5");  
} else {  
    print("$nombre est inférieur ou égal à 5");  
}
```



La condition if ... else if ... else

Permet de tester plusieurs conditions et d'exécuter un bloc de code correspondant à la première condition vraie trouvée.

Syntaxe :

```
if (condition1) {  
    // Code à exécuter si la condition1 est vraie  
} else if (condition2) {  
    // Code à exécuter si la condition2 est vraie  
} else {  
    // Code à exécuter si aucune condition n'est vraie  
    (facultatif)  
}
```

```
int note = 75;  
if (note >= 90) {  
    print("Excellent !");  
} else if (note >= 80) {  
    print("Très bien !");  
} else if (note >= 70) {  
    print("Bien !");  
} else {  
    print("Insuffisant");  
}
```



La condition switch case

Permet d'exécuter du code en fonction de la valeur d'une variable.

Syntaxe :

```
switch (expression) {  
    case valeur1:  
        // Code à exécuter si l'expression est égale à  
        valeur1 break;  
    case valeur2:  
        // Code à exécuter si l'expression est égale à  
        valeur2 break;  
    // ...default:  
        // Code à exécuter si aucune valeur ne correspond  
}
```

```
String jour = "lundi";  
switch (jour) {  
    case "lundi":  
        print("C'est lundi !");  
        break;  
    case "mardi":  
        print("C'est mardi !");  
        break;  
    // ...  
    default:  
        print("Jour inconnu");  
}
```



Déballer un optionnel

Dart dispose de plusieurs façons de déballer un optionnel.

C'est à dire le rendre nonnullable:

- En le forçant ! (nom recommandé sans vérification)
- Avec une condition if
- Avec le nullaware ??

```
String? nomOptionnel;  
late String nomIf;  
  
if (nomOptionnel != null) {  
    nomIf = nomOptionnel;  
} else {  
    nomIf = "Connais pas";  
}  
print(nomIf);  
  
String nom = nomOptionnel ?? "Inconnu";  
print(nom);
```





Les fonctions

Les fonctions

En Dart, les fonctions sont des blocs de code réutilisables qui permettent d'effectuer une tâche spécifique.

Le bloc de code ne sera effectué que si la fonction est appelée.

Elles sont essentielles pour organiser et structurer votre code, le rendre plus lisible et plus modulaire.

Syntaxe:

```
type_retour nom_fonction(paramètres) {  
    // Corps de la fonction  
}
```

Syntaxe d'appel:

```
nom_fonction(paramètres)
```

- **type_retour:** Le type de valeur que la fonction retourne. Si la fonction ne retourne aucune valeur, on utilise void.
- **nom_fonction:** Le nom unique de la fonction.
- **paramètres:** Une liste optionnelle de paramètres séparés par des virgules, entre parenthèses. Chaque paramètre a un nom et un type.
- **corps_de_la_fonction:** Le bloc de code délimité par des accolades qui contient les instructions à exécuter lorsque la fonction est appelée.



Portée des variables

Les variables déclarées dans le corps d'une fonction ont une portée locale à cette fonction.

Elles ne sont accessibles que depuis l'intérieur de la fonction et ne sont pas visibles dans le reste du programme.



Fonction sans paramètres et sans valeur de retour (procédure)

Ici la fonction ne prend aucun argument ou paramètre entre les parenthèses et ne revoie rien (void).

```
void sayHello() {  
    print("Bonjour !");  
}  
  
sayHello();
```



Fonction avec paramètres et sans valeur de retour (procédure)

Ici la fonction prend un ou plusieurs argument ou paramètre entre les parenthèses.

Chaque argument aura type suivi de son nom de variable.

Elle ne revoie rien (void).

Lors de l'appel, nous devons entrer les paramètres

```
sayHelloTo(String name) {  
    print("Bonjour $name!");  
}  
  
sayHelloTo("Matthieu");
```



Fonction avec paramètres et avec valeur de retour (procédure)

Ici la fonction prend plusieurs paramètres. ils sont séparés par des virgules.

Le type de retour est défini comme int.
Le bloc de code finit par return puis la valeur à retourner.

Rien ne sera effectué après l'expression return.

```
int additionner(int a, int b) {  
    return a + b;  
}  
  
int resultat = additionner(10, 5);  
print(resultat); // Affiche 15
```



Fonction avec arguments nommés

Ici la fonction prend plusieurs paramètres mais nommés, c'est à dire que dans l'appel de fonction, nous aurons le nom de ces paramètres.

Ils sont entre accolades et si ils ne sont pas null ou n'ont pas de valeur par défaut, il faudra ajouter le mot clé required (requis).

```
void afficherPersonne({required String nom,  
required int age}) {  
    print("Nom : $nom");  
    print("Âge : $age");  
}  
  
afficherPersonne(nom:"Bob", age: 30);
```



Fonction avec arguments par défaut

Notez dans les paramètres nommés:

- l'absence de required
- une valeur par défaut (= 35)

Si on ne passe aucun paramètre d'âge dans l'appel de fonction, la valeur par défaut sera affectée

```
void afficherPersonne({required String nom, int
age = 35}) {
    print("Nom : $nom");
    print("Âge : $age");
}

afficherPersonne(
    nom:
        "Bob"); // Affiche "Nom : Bob", "Âge : 30"
(valeur par défaut pour l'âge)
```



Fonction fléchée (lambda)

En Dart, les fonctions fléchées sont une syntaxe concise pour définir des fonctions anonymes, également connues sous le nom de fonctions lambda ou fonctions fléchées. Elles permettent d'écrire des fonctions de manière plus concise, en particulier pour les fonctions à corps simple:

(paramètres) => expression;

Les fonctions fléchées sont souvent utilisées avec des fonctions de haut niveau telles que map, where, forEach, etc.

```
int multiplier(int x, int y) => x * y;

int resultat = multiplier(6, 7);
print(resultat); // Affiche 42

var nombres = [1, 2, 3, 4, 5];
var carres = nombres.map((nombre) => nombre
* nombre);
print(carres.toList()); // Affiche: [1, 4, 9, 16, 25]
```



Fonction recursives

En Dart, une fonction réursive est une fonction qui s'appelle elle-même à l'intérieur de son propre corps. Les fonctions rékursives sont couramment utilisées pour résoudre des problèmes qui peuvent être décomposés en des sous-problèmes de taille identique ou similaire.

```
int somme(int n) {  
  if (n == 1) {  
    return 1;  
  } else {  
    print(n);  
    return n + somme(n - 1);  
  }  
}  
  
print(somme(5));
```





La programmation Orienté Objet

La programmation orienté objet

La programmation orientée objet (POO) est un paradigme de programmation qui organise le code autour de classes et d'objets. Elle permet de modéliser des concepts du monde réel ou abstrait en créant des structures de code réutilisables et maintenables.

La POO se structure autour de classes (des modèles ou des plans) pour créer des objets suivant le plan des classes

Exemple de POO:

Imaginons une application de gestion des comptes bancaires. On peut modéliser les classes suivantes:

- **Compte:** Classe représentant un compte bancaire avec des attributs comme le solde, le titulaire, etc.
- **Client:** Classe représentant un client de la banque avec des attributs comme le nom, l'adresse, etc.
- **Operation:** Classe représentant une opération bancaire comme un dépôt, un retrait, etc.



Concepts clés

- **Classe:** Un modèle ou un plan pour créer des objets. Elle définit les caractéristiques (attributs) et le comportement (méthodes) communs à tous les objets de ce type.
- **Objet:** Une instance d'une classe. Il possède les attributs et les méthodes définis par la classe et peut interagir avec d'autres objets.
- **Attribut:** Une propriété d'un objet qui stocke une valeur.
- **Méthode:** Une fonction définie dans une classe qui permet à un objet d'agir ou de réagir à une sollicitation.
- **Encapsulation:** Masquer les détails internes d'une classe et n'exposer que les méthodes et attributs publics nécessaires à son utilisation.
- **Héritage:** Permettre à une classe d'hériter des attributs et des méthodes d'une autre classe parente, favorisant la réutilisation du code.
- **Polymorphisme:** La capacité d'un objet à prendre plusieurs formes ou à se comporter de différentes manières selon le contexte.
- **Abstraction:** Se concentrer sur les aspects essentiels d'un objet ou d'une classe en masquant les détails d'implémentation complexes.



Avantages

- **Modularité:** Décomposer le code en modules indépendants et réutilisables.
- **Lisibilité:** Améliorer la clarté et la compréhension du code.
- **Maintenance:** Faciliter la modification et la correction du code.
- **Réutilisabilité:** Profiter du code existant pour créer de nouvelles fonctionnalités.
- **Extensibilité:** Adapter facilement le code à de nouveaux besoins.



La classe

En programmation orientée objet (POO), une classe est un modèle ou un plan qui définit:

- les caractéristiques (attributs)
- le comportement (méthodes) communs à un ensemble d'objets.
- On peut la considérer comme un moule à partir duquel on crée des objets individuels, qui partagent les mêmes propriétés et fonctionnalités.

Syntaxe:

```
class NomClasse {  
    // ... Attributs et méthodes  
}
```

```
class Student {  
  
    //Attributs  
  
    //Constructeur  
  
    //Méthodes  
  
}
```



L'objet

En programmation orientée objet (POO), un objet est une instance ou une matérialisation d'une classe. Il

représente une entité concrète avec ses propres caractéristiques et son propre comportement, définis par la classe à laquelle il appartient.

Création d'un objet

Pour créer un objet à partir d'une classe, on utilise un opérateur d'instanciation, généralement suivi du nom de la classe et des parenthèses contenant les valeurs des attributs à initialiser (si nécessaire)

```
void main() {  
  
    Student gerard = Student();  
  
}  
  
class Student {  
}
```



Les attributs

Les attributs, également appelés propriétés, sont des caractéristiques ou des données associées à une classe en programmation orientée objet (POO) en Dart.

Ils permettent de stocker des informations propres à chaque instance (objet) créée à partir de cette classe.

Déclaration d'attributs

Les attributs se déclarent à l'intérieur d'une classe, entre les accolades {}, en suivant la syntaxe suivante:

```
type_attribut nom_attribut;
```

```
class Student {  
  
    String firstName = "Matthieu";  
    String lastName = "Codabee";  
    int age = 34;  
    List<String> courses = ["Flutter", "Swift",  
        "Kotlin"];  
  
}
```



Accès aux attributs

On peut accéder aux attributs d'un objet en utilisant le point (.) suivi du nom de l'attribut

```
void main() {  
    Student matt = Student();  
    print(matt.age);  
}  
  
class Student {  
    String firstName = "Matthieu";  
    String lastName = "Codabee";  
    int age = 34;  
    List<String> courses = ["Flutter", "Swift",  
"Kotlin"];  
}
```



Visibilité des attributs

La visibilité des attributs (et des méthodes) est contrôlée par des conventions de nommage et par la manière dont le code est organisé en bibliothèques.

Contrairement à certains langages de programmation, Dart n'a pas de mots-clés spécifiques pour définir la visibilité publique ou privée (comme `public`, `private`, `protected` dans Java).

En Dart, pour rendre un attribut privé à la bibliothèque dans laquelle il est défini, il suffit de préfixer son nom avec un trait de soulignement (`_`). Cela signifie que l'attribut ne sera accessible qu'au sein de la même bibliothèque (généralement le même fichier Dart).

```
class Student {  
  String _firstName = "Matthieu";  
  String _lastName = "Codabee";  
  int _age = 34;  
  List<String> courses = ["Flutter", "Swift",  
    "Kotlin"];  
  
}
```



Accesseurs et mutateurs

Pour contrôler l'accès aux attributs privés, vous pouvez utiliser des accesseurs (getters) et des mutateurs (setters).

Cela permet de définir des règles spécifiques pour lire ou modifier les attributs.

L'accesseur:

```
type get nomDelAccesseur => AttributPrivé;
```

Mutateur:

```
set mutateur(Type paramètre) {  
  //code  
}
```

```
class Student {  
  String _firstName = "Matthieu";  
  String _lastName = "Codabee";  
  int _age = 34;  
  List<String> courses = ["Flutter", "Swift",  
    "Kotlin"];  
  
  String get firstName => _firstName;  
  
  set age(int nouvelAge) {  
    if (nouvelAge > _age) {  
      _age = nouvelAge;  
    }  
  }  
}
```



Le constructeur

Un constructeur est une méthode spéciale utilisée pour initialiser les objets d'une classe.

Les constructeurs permettent de configurer les objets lors de leur création.

Dart offre plusieurs types de constructeurs :

- les constructeurs par défaut
- les constructeurs nommés
- les constructeurs de redirection
- les constructeurs constants.

```
class Student {  
  String _firstName = "Matthieu";  
  String _lastName = "Codabee";  
  int _age = 34;  
  List<String> courses = ["Flutter", "Swift",  
    "Kotlin"];  
  
  String get firstName => _firstName;  
  
  set age(int nouvelAge) {  
    if (nouvelAge > _age) {  
      _age = nouvelAge;  
    }  
  }  
}
```



Le constructeur par défaut

Un constructeur par défaut est un constructeur sans paramètres qui est automatiquement appelé lorsque vous créez une instance d'une classe sans spécifier d'arguments.

Vous pouvez aussi le créer et le modifier dans votre classe

```
Student matt = Student();
```

```
//Ex de constructeur par défaut créé:
```

```
Student() {  
    _firstName = "Matt";  
}
```



Le constructeur avec paramètres

Vous pouvez définir un constructeur avec des paramètres pour initialiser les attributs avec des valeurs spécifiques lors de la création d'un objet.

Attention avec ce constructeur, les champs non initialisés devront être `late`

```
class Student {  
    late String _firstName;  
    late String _lastName;  
    int _age = 34;  
    List<String> courses = ["Flutter", "Swift", "Kotlin"];  
  
    String get firstName => _firstName;  
  
    set age(int nouvelAge) {  
        if (nouvelAge > _age) {  
            _age = nouvelAge;  
        }  
    }  
  
    Student(String first, String last) {  
        _firstName = first;  
        _lastName = last;  
    }  
}  
  
void main() {  
    Student stud = Student("Matthieu", "Codabee");  
}
```



Le constructeur Nommé

Les paramètres nommés rendent le code plus lisible et permettent de passer des arguments de manière plus flexible. La syntaxe des paramètres nommés est exactement la même que pour les fonctions.

Avec le `this` en plus

C'est le plus utilisé en Dart

```
class Student {  
  late String firstName;  
  late String lastName;  
  int age;  
  List<String> courses;  
  
  Student({required this.firstName, required this.lastName,  
    required this.age, required this.courses});  
}  
  
void main() {  
  Student stud = Student(firstName:"Matthieu", lastName:  
    "Codabee", age: 32, courses: ["Flutter"]);  
  print(stud.firstName);  
}
```



Le this

Le mot-clé `this` permet d'accéder aux attributs et méthodes de l'instance actuelle.

C'est particulièrement utile lorsque les noms des paramètres d'une méthode ou d'un constructeur sont les mêmes que les noms des attributs de la classe.

`this` peut également être utilisé pour appeler des méthodes de l'instance courante, bien que ce ne soit généralement pas nécessaire, car Dart appelle par défaut les méthodes de l'instance courante.

```
class Student {  
  late String firstName;  
  late String lastName;  
  int age;  
  List<String> courses;  
  
  Student({required this.firstName, required this.lastName,  
    required this.age, required this.courses});  
}  
  
void main() {  
  Student stud = Student(firstName:"Matthieu", lastName:  
    "Codabee", age: 32, courses: ["Flutter"]);  
  print(stud.firstName);  
}
```



Constructeur de redirection

Un constructeur de redirection appelle un autre constructeur de la même classe pour des besoins spécifiques. Ici pour recréer un nouvel arrivant

```
class Student {  
    late String firstName;  
    late String lastName;  
    int age;  
    List<String> courses;  
  
    Student(this.firstName, this.lastName, this.age, this.courses);  
  
    Student.newArrival(String first, String last) : this(first, last, 18,  
    []);  
}  
  
void main() {  
    Student stud = Student.newArrival("Matthieu", "Codabee");  
}
```



Constructeur Constant

Les constructeurs constants créent des objets immuables (qui ne peuvent pas être modifiés après leur création). Pour utiliser un constructeur constant, les variables de la classe doivent être déclarées final.

```
class Student {  
    final String firstName;  
    final String lastName;  
    final int age;  
    final List<String> courses;  
  
    const Student({required this.firstName, required  
this.lastName,required this.age,required this.courses});  
  
void main() {  
    Student stud = Student("Matthieu", "Codabee", 32, []);  
}
```



Les méthodes

En programmation orientée objet (POO), les méthodes sont des fonctions définies au sein des classes.

Elles permettent d'exécuter des opérations sur les objets et d'interagir avec les attributs de ces objets.

Les méthodes peuvent être classifiées en différentes catégories:

- Les méthodes d'instance
- Les méthodes statiques
- Les méthodes abstraites
- Les getters et les setters (nous les avons déjà vues, il s'agit des accesseurs et mutateurs).



Les méthodes d'instance

Les méthodes d'instance sont les méthodes qui opèrent sur des instances d'une classe.

Elles sont identiques aux fonctions mais au sein d'une classe.

Elles peuvent accéder et modifier les attributs de l'objet sur lequel elles sont appelées.

```
class Student {  
    String firstName;  
    String lastName;  
    int age;  
  
    Student({required this.firstName, required  
        this.lastName, required this.age});  
  
    void sayHello() {  
        print("Salut, je suis $firstName $lastName");  
    }  
}  
  
void main() {  
    Student stud = Student(firstName: "Matthieu", lastName:  
        "Codabee", age: 32);  
    stud.sayHello();  
}
```



Les méthodes statiques

Les méthodes statiques sont définies avec le mot-clé `static`.

Elles appartiennent à la classe elle-même plutôt qu'à une instance spécifique.

Elles ne peuvent pas accéder aux attributs d'instance ou aux méthodes d'instance directement, mais seulement aux attributs et méthodes statiques.

```
void main() {  
    Weather.showForecast("Marseille", 32);  
}  
  
class Weather {  
  
    static void showForecast(String city, int degrees) {  
        print("Aujourd'hui à $city il fait $degrees°C");  
    }  
}
```



Les méthodes abstraites

Les méthodes abstraites sont déclarées dans les classes abstraites (avec le mot-clé `abstract`) et n'ont pas de corps.

Elles doivent être implémentées par les sous-classes.

Nous reviendrons sur les sous classes dans la partie sur l'héritage.

```
abstract class Animal {  
    void sound();  
}  
  
class Cat extends Animal {  
  
    @override  
    void sound() {  
        print("Miaou");  
    }  
}
```



L'Héritage

L'héritage est un concept que l'on pourrait qualifier d'essentiel dans la programmation orientée objet (POO).

Il permet de créer de nouvelles classes à partir de classes existantes.

En Dart, l'héritage permet à une classe (appelée sous-classe ou classe dérivée) de hériter des attributs et des méthodes d'une autre classe (appelée superclasse ou classe de base). Un peu comme vous avez hérité des caractéristiques de vos parents.

Cela favorise la réutilisation du code et facilite la création de structures de classes plus complexes et organisées.



L'héritage: Syntaxe

1. Déclaration de la Superclasse

- `class Animal { ... }` : C'est la classe de base, qui possède un attribut `nom` et une méthode `manger`.
- Le constructeur `Animal(this.nom)` initialise l'attribut `nom`.

2. Déclaration de la Sous-classe

- `class Chien extends Animal { ... }` : C'est la sous-classe qui hérite de la classe `Animal`.
- `Chien(String nom) : super(nom);` : Le constructeur de la sous-classe appelle le constructeur de la superclasse avec `super(nom)`. Cela initialise l'attribut `nom` dans la superclasse.

3. Méthodes de la Sous-classe

- `void aboyer() { ... }` : La sous-classe peut définir ses propres méthodes en plus de celles héritées de la superclasse.

```
class Animal {  
    String nom;  
  
    Animal(this.nom);  
  
    void manger() {  
        print('$nom mange');  
    }  
}  
  
class Dog extends Animal {  
    Dog(String nom) : super(nom);  
  
    void aboyer() {  
        print('$nom aboie');  
    }  
}  
  
void main() {  
    var chien = Dog('Rex');  
    chien.manger(); // Affiche: Rex mange  
    chien.aboyer(); // Affiche: Rex aboie  
}
```



L'héritage et constructeurs

Les constructeurs de la sous-classe doivent appeler le constructeur de la superclasse pour s'assurer que les attributs hérités sont correctement initialisés.

Pour ceci on va utiliser le :super()

```
class Animal {  
    String nom;  
  
    Animal(this.nom);  
}  
  
class Chien extends Animal {  
    int age;  
  
    Chien(String nom, this.age) : super(nom);  
  
    void afficherDetails() {  
        print('Nom: $nom, Age: $age');  
    }  
}  
  
void main() {  
    var chien = Chien('Rex', 5);  
    chien.afficherDetails(); // Affiche: Nom: Rex, Age: 5  
}
```



Polymorphisme et override

Le polymorphisme (du grec "poly" : multiple et "morphe" : forme) signifie la capacité d'une même entité à prendre plusieurs formes ou à se comporter de différentes manières.

En POO, cela se traduit par la possibilité pour des objets de classes différentes de répondre au même message (appel de méthode) de manière différente.

Dans le contexte de l'héritage, le polymorphisme permet d'utiliser des instances de sous-classes à la place d'instances de la superclasse.

```
class Animal {  
    void faireDuBruit() {  
        print('L\'animal fait un bruit');  
    }  
}  
  
class Chien extends Animal {  
    @override  
    void faireDuBruit() {  
        print('Le chien aboie');  
    }  
}  
  
class Chat extends Animal {  
    @override  
    void faireDuBruit() {  
        print('Le chat miaule');  
    }  
}  
  
void main() {  
    Animal monChien = Chien();  
    Animal monChat = Chat();  
  
    monChien.faireDuBruit();  
    monChat.faireDuBruit();  
}
```



Polymorphisme et override

L'annotation `@override` en Dart est utilisée pour indiquer qu'une méthode dans une sous-classe redéfinit une méthode héritée de sa superclasse.

Cela permet à la sous-classe de fournir une implémentation spécifique de la méthode héritée.

Le polymorphisme et l'override fonctionnent souvent ensemble pour permettre un comportement spécifique tout en utilisant des interfaces ou des superclasses génériques.

Cela fonctionne aussi avec les classes abstraites

```
abstract class Animal {  
  void faireDuBruit();  
}  
  
class Chien extends Animal {  
  @override  
  void faireDuBruit() {  
    print('Le chien aboie');  
  }  
}  
  
class Chat extends Animal {  
  @override  
  void faireDuBruit() {  
    print('Le chat miaule');  
  }  
}  
  
void main() {  
  Chien monChien = Chien();  
  Chat monChat = Chat();  
  
  monChien.faireDuBruit();  
  monChat.faireDuBruit();  
}
```



Les classes Génériques

Une classe générique est une classe qui permet de travailler avec des types de données différents sans spécifier les types exacts lors de la définition de la classe.

Cela permet de créer des structures de données et des algorithmes plus flexibles et réutilisables.

Les classes génériques utilisent des paramètres de type, généralement représentés par des lettres comme T, E, K, V, etc.

Dans cet exemple, Boite est une classe générique qui accepte un paramètre de type T. Le type T peut être n'importe quel type lors de l'instanciation de la classe.

```
class Boite<T> {  
    T contenu;  
  
    Boite(this.contenu);  
  
    T getContenu() {  
        return contenu;  
    }  
  
    void setContenu(T nouveauContenu) {  
        contenu = nouveauContenu;  
    }  
}
```



Les classes Génériques

Lorsqu'on crée une instance d'une classe générique, on spécifie le type exact à utiliser pour le paramètre générique.

```
class Boite<T> {  
    T contenu;  
  
    Boite(this.contenu);  
  
    T getContenu() {  
        return contenu;  
    }  
  
    void setContenu(T nouveauContenu) {  
        contenu = nouveauContenu;  
    }  
}  
  
void main() {  
    Boite<int> boiteEntiere = Boite<int>(123);  
    print(boiteEntiere.getContenu()); // Affiche: 123  
  
    Boite<String> boiteString = Boite<String>('Bonjour');  
    print(boiteString.getContenu()); // Affiche: Bonjour  
  
    boiteString.setContenu('Au revoir');  
    print(boiteString.getContenu()); // Affiche: Au revoir  
}
```



Les classes Génériques

Lorsqu'on crée une instance d'une classe générique, on spécifie le type exact à utiliser pour le paramètre générique.

```
class ErrCode<K, V> {  
    K cle;  
    V valeur;  
  
    ErrCode(this.cle, this.valeur);  
  
    K getCle() {  
        return cle;  
    }  
  
    V getValeur() {  
        return valeur;  
    }  
}  
  
void main() {  
    ErrCode<int, String> paire = ErrCode<int, String>(404, "Page not  
found");  
    print('Code: ${paire.getCle()}, Erreur: ${paire.getValeur()}');  
    // Affiche: Clé: Un, Valeur: 1  
}
```



Les classes Génériques

Avantages des Classes Génériques

1. **Réutilisabilité du Code** : Les classes génériques permettent d'écrire des algorithmes et des structures de données qui fonctionnent avec n'importe quel type de données.
2. **Sécurité de Type** : En spécifiant les types lors de l'instanciation, on obtient des vérifications de type à la compilation, réduisant ainsi les erreurs liées aux types.
3. **Flexibilité** : Les classes génériques peuvent être utilisées avec des types personnalisés, ce qui les rend très flexibles et adaptées à une large gamme de situations.



Les classes Génériques

Avantages des Classes Génériques

1. Réutilisabilité du Code : Les classes génériques permettent d'écrire des algorithmes et des structures de données qui fonctionnent avec n'importe quel type de données.
2. Sécurité de Type : En spécifiant les types lors de l'instanciation, on obtient des vérifications de type à la compilation, réduisant ainsi les erreurs liées aux types.
3. Flexibilité : Les classes génériques peuvent être utilisées avec des types personnalisés, ce qui les rend très flexibles et adaptées à une large gamme de situations.

```
class ErrCode<K, V> {
    K cle;
    V valeur;

    ErrCode(this.cle, this.valeur);

    K getCle() {
        return cle;
    }

    V getValeur() {
        return valeur;
    }
}

void main() {
    ErrCode<int, String> paire = ErrCode<int, String>(404, "Page not found");
    print('Code: ${paire.getCle()}, Erreur: ${paire.getValeur()}');
    // Affiche: Clé: Un, Valeur: 1
}
```



Les Enum

Les énumérations, ou enums en anglais, sont une fonctionnalité de nombreux langages de programmation, y compris Dart.

Ils permettent de définir un type spécial qui représente un ensemble fixe de constantes nommées.

Ces constantes sont souvent utilisées pour représenter des choix, des états ou des options prédéfinies dans un programme.



Enum

En Dart, on définit une énumération à l'aide du mot-clé `enum`

Puis nous énumérons les cas possibles séparés par des virgules

```
enum Jour {  
  lundi,  
  mardi,  
  mercredi,  
  jeudi,  
  vendredi,  
  samedi,  
  dimanche  
}
```



Utilisation des Enums

Une fois qu'une énumération est définie, on peut l'utiliser pour déclarer des variables, des paramètres de méthode, ou des valeurs de retour de fonction :

```
enum Jour {  
    lundi,  
    mardi,  
    mercredi,  
    jeudi,  
    vendredi,  
    samedi,  
    dimanche;  
}  
  
void main() {  
    var aujourdHui = Jour.mardi;  
  
    void afficherJour(Jour jour) {  
        switch (jour) {  
            case Jour.lundi:  
                print("C'est lundi");  
                break;  
            case Jour.mardi:  
                print("C'est mardi");  
                break;  
            // Les autres cas...  
        }  
    }  
  
    afficherJour(aujourdHui); // Affiche: C'est mardi  
}
```

