

Introduction au Dart



Le langage de programmation Dart a été développé par Google en 2010 et lancé la même année. Dart est conçu pour être une alternative à JavaScript et est utilisé principalement pour le développement d'applications web et mobiles.

Table des matières

1. Introduction au Dart	3
2. Les variables en Dart	5
3. Structures de contrôle en Dart	6
4. Les fonctions en Dart	8
5. Collections de données en Dart	10
6. Classes et objets en Dart	14
7-préférez_initialiser_les_formels	15
Détails	15
Usage	17
7. Gestion des erreurs en Dart	19
8. Programmation asynchrone en Dart	21
9-Débogage des applications Web Dart	22
Aperçu	22
Premiers pas avec Dart DevTools.....	23
Obtention de packages d'outils de ligne de commande.....	25
Débogage du code de production.....	26
Chrome	26
Bord	26
Firefox	26
Safari	27
Ressources	27
Conclusion	27

1. Introduction au Dart

Qu'est-ce que le langage Dart ?

Dart est un langage de programmation, qui a été et est développé principalement par Google. Il est standardisé par Ecma (Ecma est une organisation internationale de normalisation des systèmes d'information et de communication ainsi que des appareils électroniques grand public). La programmation dans Dart doit être une **alternative attrayante à JavaScript** dans des navigateurs Web modernes. D'après les développeurs de Dart, les faiblesses de JavaScript ne peuvent plus être corrigées par son développement.

Le langage Dart est développé depuis 2010 et lancé la même année. Comme les navigateurs ne pouvaient et ne peuvent pas le manier en mode natif et que JavaScript peut être exécuté dans tous les navigateurs modernes, il existe le **Compiler Dart2js**, autrement dit « Dart vers JavaScript ». Le langage Dart ressemble aux **langages de programmation orientés objet** établis, dont Swift, C# ou Java font partie, qui sont soumis à des paradigmes de programmation déterminés. Les règles de combinaison de signes établis, c'est-à-dire la **syntaxe**, ressemblent au langage de programmation C. Cette « parenté » facilite grandement la prise en main de sorte que l'on peut y accéder sans avoir à surmonter trop d'obstacles.

Comment Dart est-il construit ?

Le langage Dart dispose de variables, d'opérateurs, de directives conditionnées, de boucles, de fonctions, de catégories, d'objets et d'énumérations. Il offre la transmission et la programmation générique comme concepts importants d'un langage orienté objet, de nombreux éléments qu'un programmeur expérimenté connaît déjà. Pour un premier essai, la plateforme open source gratuite [DartPad](#) est disponible, elle permet de se familiariser avec la programmation en langage Dart et propose quelques **exemples** sous la forme d'un menu déroulant.

Tout programme écrit en Dart commence par l'appel de la fonction « Main ».

```
void main() {  
  
}
```

Copy

Comme exemple, l'établissement d'une variable et l'exécution d'une requête de condition :

```
void main() {
```

```
var animal = 'horse';

if (animal == 'cat' || animal == 'dog') {

    print('This animal is a pet.');
```



```
    } else if (animal == 'elephant') {

        print('That\'s a huge animal.');
```



```
    } else {

        print('This animal is not a pet.');
```



```
    }

}
```

Lien :

[Dart | Le langage de programmation de Google illustré par des exemples - IONOS](#)

Présentation de Dart:

Le Dart se distingue par sa **syntaxe claire et lisible**, adaptée aux débutants comme aux experts. Sa polyvalence le rend idéal pour créer diverses applications, allant des **applications web aux mobiles** et aux applications de bureau.

Avantages et usages:

Le Dart est reconnu pour sa **rapidité d'exécution** grâce à son moteur JIT et AOT. C'est aussi le langage phare de **Flutter**, idéal pour créer des interfaces fluides et interactives.

Installation et configuration:

Pour commencer, **installez le Dart SDK**, puis configurez votre environnement de développement (IDE). Choisissez parmi des options comme **Visual Studio Code**, IntelliJ IDEA ou Android Studio. Prêt à plonger dans la programmation Dart ? Commencez le programme gratuit de Flutter Révolution pour accéder à toutes ces étapes d'installation de Dart et Flutter.

2. Les variables en Dart

Variables et types de données en programmation

Les variables jouent un **rôle fondamental en programmation** en permettant aux développeurs de stocker et de manipuler des données de manière dynamique. Une variable peut être considérée comme une "boîte" où vous pouvez **stocker différentes valeurs**, et ces valeurs peuvent changer au cours de l'exécution du programme. Les variables permettent de rendre les programmes plus flexibles et réutilisables en manipulant les données de manière abstraite.

Types de données de base en Dart

[Le langage de programmation Dart a été développé par Google en 2010 et lancé la même année.](#) Dart est conçu pour être une alternative à JavaScript et est utilisé principalement pour le développement d'applications web et mobiles., il existe plusieurs **types de données de base**:

1. **Entiers (int)**: Les entiers représentent des nombres entiers sans partie décimale. Ils peuvent être positifs, négatifs ou nuls. Par exemple, 5, -10 et 0 sont des entiers en Dart.
2. **Décimaux (double)**: Les décimaux sont utilisés pour représenter des nombres avec une partie décimale. Ils peuvent également représenter des nombres entiers. Par exemple, 3.14, -0.5 et 10.0 sont des décimaux en Dart.
3. **Chaînes de caractères (String)**: Les chaînes de caractères sont utilisées pour représenter du texte. Elles sont entourées de guillemets simples (") ou doubles (""). Par exemple, "Hello, world !" est une chaîne de caractères en Dart.
4. **Booléens (bool)**: Les booléens sont utilisés pour représenter des valeurs binaires, généralement vrai `true` ou faux `false`. Ils sont essentiels pour les structures de contrôle, telles que les instructions conditionnelles. Par exemple, true et false sont des valeurs booléennes en Dart.

Déclaration et utilisation des variables en Dart

Pour déclarer une variable en Dart, vous devez **spécifier son type** et lui attribuer une **valeur initiale**. Voilà comment on fait en Dart

```
// Déclaration d'une variable entière
int age = 27;

// Déclaration d'une variable décimale
double prix = 19.99;

// Déclaration d'une variable chaîne de caractères
```

```
String nom = "Waffo";

// Déclaration d'une variable booléenne
bool estMajeur = true;
```

Une fois déclarées, vous pouvez **utiliser ces variables** dans votre code pour effectuer des opérations, des comparaisons ou les afficher à l'écran:

```
// Utilisation des variables
print("Bonjour, je m'appelle $nom et j'ai $age ans.");
if (estMajeur) {
    print("Je suis majeur(e) !");
} else {
    print("Je ne suis pas encore majeur(e).");
}
```

En résumé, les variables sont essentielles pour **stocker et manipuler des données** en programmation. En Dart, vous pouvez utiliser différents types de données de base tels que les entiers, les décimaux, les chaînes de caractères et les booléens. La déclaration et l'utilisation de variables en Dart vous permettent de créer des **programmes dynamiques** et interactifs.

3. Structures de contrôle en Dart

Structures de contrôle en programmation

Les structures de contrôle sont des éléments essentiels en programmation qui permettent de **gérer le flux d'exécution** d'un programme. Elles permettent de prendre des **décisions conditionnelles** et d'itérer sur des données à l'aide de boucles. Voici comment elles fonctionnent :

Utilisation des structures conditionnelles (if, else, else if)

Les structures conditionnelles permettent **d'exécuter** différentes **parties de code** en fonction de **conditions spécifiques**. En Dart, les structures conditionnelles les plus couramment utilisées sont les suivantes :

1. **if**: Exécute un bloc de code si la **condition est vraie** (true).

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

```
}
```

2. **else if**: Permet de tester **plusieurs conditions** en série.

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

Utilisation des boucles (for, while) pour itérer sur des données

Les boucles permettent **d'exécuter un bloc de code plusieurs fois**, en fonction d'une condition. En Dart, les boucles les plus courantes sont les suivantes:

1. **for**: Utilisé pour itérer un nombre spécifique de fois.

```
for (int i = 0; i < 5; i++) {  
    // Code à exécuter à chaque itération  
}
```

2. **while**: Utilisé pour itérer tant qu'une condition est vraie.

```
3. while (condition) {  
4.     // Code à exécuter tant que la condition est vraie  
5. }
```

Utilisation des structures de contrôle pour gérer le flux d'exécution du programme

Les structures de contrôle permettent de **gérer le flux d'exécution** d'un programme en prenant des décisions et en contrôlant les itérations. Elles sont essentielles pour rendre les programmes dynamiques et interactifs. Par exemple, vous pouvez utiliser une combinaison de structures conditionnelles et de boucles pour gérer différentes situations, telles que des **interactions utilisateur**, des calculs complexes, etc.

Voici un exemple **d'utilisation combinée** de structures conditionnelles et de boucles pour illustrer le concept :

```
int nombre = 10;

if (nombre > 0) {
    for (int i = 0; i < nombre; i++) {
        print("Itération $i");
    }
} else {
    print("Le nombre est négatif ou nul.");
}
```

Dans cet exemple, le programme affiche une série de messages d'itération **si le nombre est positif**, sinon il affiche un message indiquant que le nombre est **négatif ou nul**.

En résumé, les structures de contrôle en programmation, telles que les structures conditionnelles (if, else, else if) et les **boucles** (for, while), sont utilisées pour **prendre des décisions** et **itérer** sur des données. Elles permettent de contrôler le flux d'exécution du programme et de créer des logiques complexes et interactives.

4. Les fonctions en Dart

Fonctions en programmation Dart

Les fonctions sont des **blocs de code réutilisables** qui permettent d'organiser et de structurer un programme en effectuant une tâche spécifique. Elles favorisent la modularité et la **réutilisabilité du code** en le divisant en petites parties autonomes. En Dart, les fonctions jouent un rôle essentiel dans le développement logiciel.

Déclaration et utilisation des fonctions

Pour déclarer une fonction en Dart, vous spécifiez son **nom**, les éventuels **paramètres** qu'elle prend et le **type de valeur** qu'elle renvoie (si elle renvoie une valeur). Voici la **syntaxe de base**:

```
typeDeRenvoi nomDeLaFonction(paramètres) {
    // Code à exécuter
    // ...
    return valeur; // (si la fonction renvoie une valeur)
}
```

Pour utiliser une fonction, vous **appelez** simplement **son nom** suivi des parenthèses, en fournissant les arguments nécessaires si la fonction prend des paramètres.

Passage de paramètres et retour de valeurs

Les **paramètres** sont des valeurs que vous **passez à une fonction** pour qu'elle les utilise dans son traitement. Il existe **deux types** de paramètres en Dart: les paramètres requis et les paramètres optionnels.

1. **Paramètres requis** : Ces paramètres doivent être fournis lors de l'appel de la fonction.

```
void hello(String nom) {  
    print("Bonjour, $nom !");  
}  
  
// Appel de la fonction  
hello("Waffo"); // Affiche : Bonjour, Waffo !
```

2. **Paramètres optionnels** : Ces paramètres peuvent être omis lors de l'appel de la fonction, et vous pouvez définir une valeur par défaut.

```
void afficherDetails(String nom, {int age = 0, String pays = "Inconnu"}) {  
    print("Nom: $nom, Age: $age, Pays: $pays");  
}  
  
// Appel de la fonction  
afficherDetails("Waffo", age: 27); // Affiche : Nom: Waffo, Age: 27, Pays: Inconnu
```

Pour **renvoyer une valeur** à partir d'une fonction, utilisez le mot-clé **return**.

```
int somme(int a, int b) {  
    return a + b;  
}  
  
// Appel de la fonction  
int resultat = somme(5, 3);  
print("La somme est : $resultat"); // Affiche : La somme est : 8
```

En résumé, les fonctions en Dart sont des **blocs de code réutilisables** qui effectuent des tâches spécifiques. Vous pouvez définir des fonctions avec des paramètres requis ou optionnels, ainsi qu'avec ou sans valeur de renvoi. Les fonctions favorisent la modularité, la **réutilisabilité** et la **clarté du code**.

5. Collections de données en Dart

Collections de données en Dart

En Dart, les collections de données sont des structures qui vous permettent de stocker et de **gérer plusieurs valeurs** de manière organisée. Les **trois types** de collections de données les plus courants sont les listes, les ensembles et les cartes.

1. **Listes:** Les listes sont des collections ordonnées d'éléments, où chaque élément peut être répété et identifié par son index. Elles permettent d'accéder aux éléments en fonction de leur position dans la liste.
2. **Ensembles:** Les ensembles sont des collections non ordonnées d'éléments uniques. Ils garantissent que chaque élément ne peut apparaître qu'une seule fois dans l'ensemble.
3. **Maps:** Les maps sont des collections d'éléments associés à des clés uniques. Chaque élément est stocké avec une clé qui permet de le retrouver rapidement.

Manipulation des listes:

- *Création d'une liste:*

```
List<int> nombres = [1, 2, 3, 4, 5];
```

- *Ajout d'éléments à la fin de la liste:*

```
nombres.add(6);
```

- *Suppression d'un élément:*

```
nombres.remove(3); // Supprime l'élément 3
```

- *Recherche d'un élément:*

```
int index = nombres.indexOf(4); // Donne l'index de l'élément 4
```

Manipulation des ensembles:

- *Création d'un ensemble:*

```
Set<String> pays = {"USA", "Canada", "France"};
```

- *Ajout d'éléments:*

```
pays.add("Allemagne");
```

- *Suppression d'éléments:*

```
pays.remove("Canada");
```

Manipulation d'une Map:

- **Création d'une Map:**

```
Map<String, String> capitales = {  
    "France": "Paris",  
    "Allemagne": "Berlin",  
    "Espagne": "Madrid"  
};
```

- **Ajout d'éléments:**

```
capitales["Italie"] = "Rome";
```

- **Suppression d'éléments:**

```
capitales.remove("Allemagne");
```

Création et Initialisation d'une Map

Dart

```
void main() {  
  
    // Création d'une Map avec des noms et des âges  
  
    Map<String, int> personnes = {  
  
        'Waffo': 30,  
  
        'Eve': 25,  
  
        'Myra': 28  
  
    };  
  
  
  
    print(personnes); // Affiche: {Waffo: 30, Eve: 25, Myra: 28}  
  
}
```

Ajout d'un Élément

Dart

```
void main() {  
  
  Map<String, int> personnes = {  
  
    'Waffo': 30,  
  
    'Eve': 25,  
  
    'Myra': 28  
  
  };  
  
  
  // Ajout d'un nouvel élément  
  
  personnes['John'] = 35;  
  
  
  print(personnes); // Affiche: {Waffo: 30, Eve: 25, Myra: 28, John: 35}  
  
}
```

Mise à Jour d'un Élément

Dart

```
void main() {  
  
  Map<String, int> personnes = {  
  
    'Waffo': 30,  
  
    'Eve': 25,  
  
    'Myra': 28  
  
  };  
  

```

```
// Mise à jour de l'âge de Myra

personnes['Myra'] = 29;

print(personnes); // Affiche: {Waffo: 30, Eve: 25, Myra: 29}

}
```

Suppression d'un Élément

Dart

```
void main() {

  Map<String, int> personnes = {

    'Waffo': 30,

    'Eve': 25,

    'Myra': 28

  };

  // Suppression de l'élément 'Eve'

  personnes.remove('Eve');

  print(personnes); // Affiche: {Waffo: 30, Myra: 28}

}
```

Itération sur une Map

Dart

```
void main() {  
  
  Map<String, int> personnes = {  
  
    'Waffo': 30,  
  
    'Eve': 25,  
  
    'Myra': 28  
  
  };  
  
  
  // Itération sur les éléments de la Map  
  
  personnes.forEach((nom, age) {  
  
    print('$nom a $age ans');  
  
  });  
  
  // Affiche:  
  
  // Waffo a 30 ans  
  
  // Eve a 25 ans  
  
  // Myra a 28 ans  
  
}
```

résumé, les collections de données en Dart (listes, ensembles et cartes) offrent des moyens flexibles et puissants pour stocker et gérer des données de **différentes manières**. Vous pouvez ajouter, supprimer et rechercher des éléments dans ces collections, ce qui facilite la manipulation des données dans vos programmes.

6. Classes et objets en Dart

Attention: Mise à jour du code source dans cette partie, pour la syntaxe du constructeur, pour plus d'informations:

7-préferer_initialiser_les_formels

1. [Outils](#) chevron_droit
2. [Règles de Linter](#) chevron_droit
3. [prefer_initializing_formals](#)

Utilisez des formules d'initialisation formelles lorsque cela est possible.

Cette règle est disponible à partir de Dart 2.0.0.

Ensembles de règles : [recommandé](#), [flutter](#)

Cette règle dispose d'une [solution rapide](#).

Détails

#

Utilisez des formules d'initialisation formelles lorsque cela est possible.

L'utilisation de formalités d'initialisation lorsque cela est possible rend votre code plus concis.

MAUVAIS:

```
dard
class Point {
  num x, y;
  Point(num x, num y) {
    this.x = x;
    this.y = y;
  }
}
```

BIEN:

```
dard
```

```
class Point {
  num x, y;
  Point(this.x, this.y);
}
```

MAUVAIS:

dard

```
class Point {
  num x, y;
  Point({num x, num y}) {
    this.x = x;
    this.y = y;
  }
}
```

BIEN:

dard

```
class Point {
  num x, y;
  Point({this.x, this.y});
}
```

REMARQUE : cette règle ne génère pas de lint pour les paramètres nommés, sauf si le nom du paramètre et le nom du champ sont identiques. La raison en est que la résolution d'un tel lint nécessiterait soit de renommer le champ, soit de renommer le paramètre, et ces deux actions pourraient potentiellement constituer une modification radicale. Par exemple, ce qui suit ne générera pas de lint :

dard

```
class Point {
  bool isEnabled;
  Point({bool enabled}) {
```



```

    this.isEnabled = enabled; // OK
  }
}

```

REMARQUE : Notez également qu'il est possible d'appliquer un type plus strict que le champ initialisé avec un paramètre formel d'initialisation. Dans l'exemple suivant, le `Bid` constructeur sans nom requiert une valeur non nulle `int` bien qu'il `amount` soit déclaré nullable (`int?`).

dart

```

class Bid {
  final int? amount;
  Bid(int this.amount);
  Bid.pass() : amount = null;
}

```

Usage

#

Pour activer la `prefer_initializing_formals` règle, ajoutez `prefer_initializing_formals` sous **linter > règles** dans votre `analysis_options.yaml` fichier :

options_d'analyse.yaml

YAML

```

linter:
  rules:
    - prefer_initializing_formals

```

Classes et objets en programmation orientée objet en Dart

La programmation orientée objet (POO) est un paradigme de programmation qui repose sur la **création de classes et d'objets** pour organiser et structurer le code de manière plus modulaire et orientée vers les concepts du monde réel. En Dart, la POO est largement utilisée pour créer des systèmes complexes et interconnectés.

Déclaration de classes et création d'objets

Une classe est un modèle pour créer des objets. Elle définit les propriétés (**attributs**) et les comportements (**méthodes**) que les objets auront. Voici comment déclarer **une classe en Dart**:

```
class Personne {  
    String nom;  
    int age;  
  
    // Constructeur  
    Personne(this.nom, this.age);  
}
```

Pour **créer un objet** à partir de cette classe, utilisez le constructeur :

```
Personne personne1 = Personne("Waffo", 27);  
Personne personne2 = Personne("Eva", 30);
```

Utilisation des propriétés et des méthodes des objets

Les propriétés (**attributs**) d'un objet sont des **variables** associées à cet objet, tandis que les **méthodes** sont des **fonctions** associées à cet objet. Vous pouvez accéder aux propriétés et appeler les méthodes d'un objet de la manière suivante:

```
print(personne1.nom); // Affiche : Waffo  
print(personne2.age); // Affiche : 30  
  
class Personne {  
    String nom;  
    int age;  
  
    // Constructeur  
    Personne(this.nom, this.age);  
  
    // Méthode  
    void sePresenter() {  
        print("Bonjour, je m'appelle $nom et j'ai $age ans.");  
    }  
}  
  
personne1.sePresenter(); // Affiche : Bonjour, je m'appelle Waffo et j'ai 27 ans.  
personne2.sePresenter(); // Affiche : Bonjour, je m'appelle Eva et j'ai 30 ans.
```

Encapsulation et méthodes d'accès

En Dart, vous pouvez définir des méthodes d'accès (**getters** et **setters**) pour contrôler l'accès aux propriétés privées d'une classe et pour encapsuler la logique d'accès. Par exemple:

```
class CompteBancaire {  
    double _solde = 0;  
  
    // Méthode getter  
    double get solde {  
        return _solde;  
    }  
  
    // Méthode setter  
    set solde(double montant) {  
        if (montant >= 0) {  
            _solde = montant;  
        }  
    }  
}  
  
CompteBancaire compte = CompteBancaire();  
compte.solde = 1000; // Utilisation du setter  
print(compte.solde); // Utilisation du getter
```

En résumé, la programmation orientée objet en Dart repose sur **les classes et les objets**. Les classes définissent les propriétés et les méthodes, et les objets sont des instances de ces classes. Vous pouvez créer, manipuler et **interagir avec des objets** en utilisant leurs propriétés et leurs méthodes. L'encapsulation permet de contrôler l'accès aux **propriétés privées**, et les méthodes d'accès (getters et setters) peuvent être utilisées pour encapsuler la logique d'accès.

7. Gestion des erreurs en Dart

Gestion des erreurs en Dart

La gestion des erreurs est une **partie cruciale de la programmation**, car elle permet de détecter, signaler et gérer les problèmes qui peuvent survenir lors de l'exécution d'un programme. En Dart, les erreurs sont gérées à l'aide de mécanismes de **gestion d'exceptions**.

Utilisation des blocs try-catch pour capturer et gérer les exceptions

Les exceptions sont des **situations anormales** ou des **erreurs** qui se produisent pendant l'exécution d'un programme. Les blocs **try** et **catch** sont utilisés pour gérer les exceptions en Dart:

```
try {  
    // Code susceptible de générer une exception  
} catch (exception) {  
    // Code de gestion de l'exception  
}
```

Dans ce bloc, le code à l'intérieur du bloc **try** est exécuté. Si **une exception est repérée** pendant cette exécution, le code à l'intérieur du bloc **catch** est exécuté pour gérer l'exception.

```
try {  
    var resultat = 10 ~/ 0; // Division par zéro  
} catch (e) {  
    print("Une exception s'est produite : $e");  
}
```

Utilisation des exceptions personnalisées

En plus des **exceptions intégrées**, vous pouvez également créer vos propres exceptions **personnalisées** en définissant des classes qui héritent de la classe **Exception** ou d'autres classes liées aux exceptions. Cela peut être utile lorsque vous souhaitez gérer des **situations spécifiques** de manière plus détaillée.

```
class MonException implements Exception {  
    String message;  
  
    MonException(this.message);  
  
    @override  
    String toString() {  
        return "MonException : $message";  
    }  
}  
  
void exemple() {  
    try {  
        throw MonException("Ceci est mon exception personnalisée");  
    } catch (e) {  
        print(e);  
    }  
}
```

Dans cet exemple, une exception personnalisée `MonException` est créée en implémentant l'interface `Exception`. Lorsque cette exception est lancée et capturée, son message spécifié est affiché. En résumé, la **gestion des erreurs** en Dart est réalisée à l'aide des blocs `try` et `catch`, où le code pouvant générer des exceptions est placé dans le bloc `try`, et le code pour gérer les exceptions est placé dans le bloc `catch`. Vous pouvez également créer des **exceptions personnalisées** pour gérer des situations spécifiques de manière plus précise.

8. Programmation asynchrone en Dart

Programmation asynchrone en Dart

La programmation asynchrone est utilisée pour gérer des **opérations qui prennent du temps à s'exécuter**, comme les appels réseau, les accès à la base de données ou d'autres opérations E/S. Elle permet au programme de continuer à s'exécuter pendant que ces opérations sont en cours, évitant ainsi de bloquer le thread principal et améliorant la **réactivité de l'application**.

Utilisation des futures et des promesses pour gérer les opérations asynchrones

En Dart, les futures et les promesses sont utilisées pour gérer les opérations asynchrones. Un **futur** est une référence à une valeur qui peut ne **pas être encore disponible**, tandis qu'une **promesse** est l'objet qui **produit cette valeur** à un moment donné.

```
Future<String> obtenirDonnees() {  
    return Future.delayed(Duration(seconds: 2), () => "Données reçues");  
}
```

Pour utiliser le résultat d'un **futur**, vous pouvez utiliser la méthode `.then()` :

```
obtenirDonnees().then((resultat) {  
    print(resultat); // Affiche : Données reçues  
});
```

Utilisation de l'opérateur `async/await` pour simplifier la programmation asynchrone

L'opérateur `async` est utilisé pour marquer une **fonction** comme **asynchrone**, ce qui signifie qu'elle peut contenir des opérations asynchrones. L'opérateur `await` est utilisé à l'intérieur de fonctions asynchrones pour attendre la résolution d'un futur ou d'une promesse.

```
Future<void> exempleAsynchrone() async {  
    try {  
        var resultat = await obtenirDonnees();  
        print(resultat); // Affiche : Données reçues  
    } catch (erreur) {  
        print("Une erreur s'est produite : $erreur");  
    }  
}
```

Dans cet exemple, `await obtenirDonnees()` attend que la fonction `obtenirDonnees()` renvoie une valeur avant de continuer. Si une exception est levée dans le futur, elle est **capturée et gérée** dans le bloc `catch`.

L'opérateur `async/await` rend la programmation asynchrone **plus lisible** et plus semblable à la programmation synchrone, ce qui facilite la gestion des **opérations asynchrones** dans votre code. En résumé, la programmation asynchrone en Dart est utilisée pour gérer les **opérations qui prennent du temps** à s'exécuter. Les futures et les promesses sont utilisées pour représenter les valeurs futures, tandis que les opérateurs `async` et `await` simplifient la gestion des opérations asynchrones en les rendant plus lisibles et similaires à la programmation synchrone.

9-Débogage des applications Web Dart

Vous pouvez utiliser un [IDE Dart](#), [Dart DevTools](#) et des outils de navigateur tels que [Chrome DevTools](#) pour déboguer vos applications Web Dart.

- Pour déboguer la logique de votre application, utilisez votre IDE, Dart DevTools ou les outils de votre navigateur. Dart DevTools offre une meilleure prise en charge que les outils de navigateur pour inspecter et recharger automatiquement le code Dart.
- Pour déboguer l'apparence (HTML/CSS) et les performances de votre application, utilisez votre IDE ou des outils de navigateur tels que Chrome DevTools.

Aperçu

#

Pour servir votre application, utilisez `webdev serve`(soit sur la ligne de commande, soit via votre IDE) pour démarrer le compilateur de développement Dart. Pour activer Dart DevTools, ajoutez l'option `--debug` ou `--debug-extension`(sur la ligne de commande ou via votre IDE) :

```
$ webdev serve --debug
```

contenu_copie

Lorsque vous exécutez votre application à l'aide `--debug` de l'indicateur `webdev`, vous pouvez ouvrir Dart DevTools en appuyant sur **Alt+D**(ou **Option+D** sur macOS).

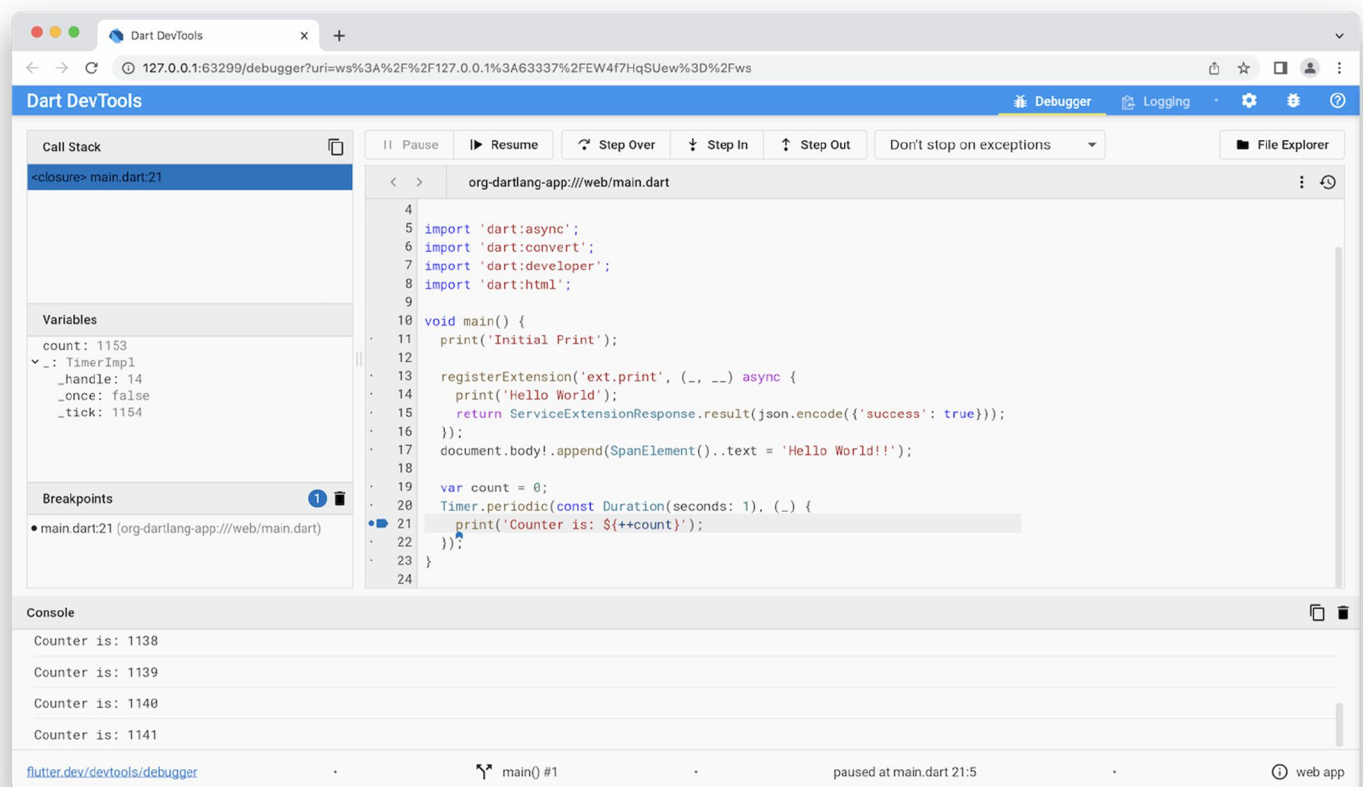
Pour ouvrir Chrome DevTools, appuyez sur **Control+Shift+I**(ou **Command+Option+I** sur macOS). Si vous souhaitez déboguer votre application à l'aide de Chrome DevTools, vous pouvez utiliser [des cartes source](#) pour afficher vos fichiers source Dart au lieu du JavaScript produit par le compilateur. Pour plus d'informations sur l'utilisation de Chrome DevTools, consultez la [documentation de Chrome DevTools](#).

Pour utiliser Dart DevTools ou Chrome DevTools pour déboguer une application Web Dart, vous avez besoin du logiciel suivant :

- [Google Chrome](#).
- [Dart SDK](#), version 2.0.0 ou supérieure.
- L'un des environnements de développement suivants :
 - Ligne de commande : [packages d'outils de ligne de commande Dart](#) tels que webdev (requis pour Dart et Chrome DevTools) et devtools (requis pour Dart DevTools).
 - ou
 - Un [IDE ou un éditeur Dart](#) qui prend en charge le développement Web.
- Une [application Web Dart](#) à déboguer.

Premiers pas avec Dart DevTools

#



Cette section vous guide à travers les bases de l'utilisation de Dart DevTools pour déboguer une application Web. Si vous disposez déjà d'une application prête à être déboguée, vous pouvez ignorer la création de l'application de test (étape 1), mais vous devrez ajuster les instructions pour qu'elles correspondent à votre application.

1. *Facultatif* : clonez le [référentiel WebDev](#) afin de pouvoir utiliser son exemple d'application pour jouer avec Dart DevTools.
2. *Facultatif* : installez l' [extension Dart Debug](#) afin de pouvoir exécuter votre application et ouvrir Dart DevTools dans une instance déjà en cours d'exécution de Chrome.

3. Dans le répertoire principal de votre application, exécutez `dart pub get` pour obtenir ses dépendances.

```
4. $ cd example
   $ dart pub get
```

contenu_copie

5. Compilez et diffusez l'application en mode débogage, en utilisant soit votre IDE, soit `webdev` la ligne de commande.

informationsNote

La première compilation est celle qui prend le plus de temps, car l'application entière doit être compilée. Ensuite, les actualisations sont beaucoup plus rapides.

Si vous utilisez `webdev` sur la ligne de commande, la commande à utiliser dépend de si vous souhaitez (ou devez) exécuter l'application et le débogueur dans une instance déjà en cours d'exécution de Chrome.

- Si vous avez installé [l'extension Dart Debug](#) et que vous souhaitez utiliser une instance existante de Chrome pour déboguer :

```
$ webdev serve --debug-extension
```

contenu_copie

- Sinon, utilisez la commande suivante, qui lance une nouvelle instance de Chrome et exécute l'application :

```
$ webdev serve --debug
```


contenu_copie

6. Si votre application n'est pas déjà en cours d'exécution, ouvrez-la dans une fenêtre de navigateur Chrome.

Par exemple, si vous utilisez `webdev serve --debug-extensions` sans argument, ouvrez <http://127.0.0.1:8080> .

7. Ouvrez Dart DevTools pour déboguer l'application en cours d'exécution dans la fenêtre actuelle.

- Si l'extension Dart Debug est installée et que vous avez utilisé l' `--debug-`

`extension` indicateur pour `webdev`, cliquez sur le logo Dart  en haut à droite de la fenêtre du navigateur.



- Si vous avez utilisé le `--debug` drapeau pour `webdev`, appuyez sur **Alt+D** (ou **Option+D** sur macOS).

La fenêtre Dart DevTools s'ouvre et affiche le code source du fichier principal de votre application.

8. Définissez un point d'arrêt à l'intérieur d'un minuteur ou d'un gestionnaire d'événements en cliquant à gauche de l'une de ses lignes de code.
Par exemple, cliquez sur le numéro de ligne de la première ligne à l'intérieur d'un gestionnaire d'événements ou d'un rappel de minuteur.
9. Déclenchez l'événement qui provoque l'appel de fonction. L'exécution s'arrête au point d'arrêt.
10. Dans le volet **Variables**, inspectez les valeurs des variables.
11. Reprenez l'exécution du script et déclenchez à nouveau l'événement ou appuyez sur **Pause**. L'exécution s'interrompt à nouveau.
12. Essayez de parcourir le code ligne par ligne à l'aide des boutons **Entrer**, **Passer au-dessus** et **Sortir**.

informationsNote

Dart DevTools n'intervient pas dans le code du SDK. Par exemple, si vous appuyez sur **Step In** lors d'un appel à `print()`, vous passez à la ligne suivante, et non dans le code du SDK qui implémente `print()`.

13. Modifiez votre code source et rechargez la fenêtre Chrome qui exécute l'application.
L'application se reconstruit et se recharge rapidement. Tant que [le problème 1925](#) n'est pas résolu, vous perdez vos points d'arrêt lors du rechargement de l'application.
14. Cliquez sur le bouton **Journalisation** pour voir les journaux stdout, stderr et système.

Obtention de packages d'outils de ligne de commande

#

Si vous utilisez la ligne de commande au lieu d'un IDE ou d'un éditeur compatible Dart, vous avez besoin de l' [outil webdev](#). Dart DevTools est fourni par le SDK.

```
$ dart pub global activate webdev
```

contenu_copie

Si votre variable d'environnement PATH est correctement configurée, vous pouvez désormais utiliser ces outils sur la ligne de commande :

```
$ webdev --help
A tool to develop Dart web projects.
...
```

contenu_copie

Pour plus d'informations sur la définition de PATH, consultez la [dart pub global documentation](#).

Chaque fois que vous mettez à jour le SDK Dart, mettez à jour les outils en les activant à nouveau :

```
$ dart pub global activate webdev # update webdev
```

contenu_copie

Débogage du code de production

#

Cette section fournit des conseils pour déboguer le code compilé en production dans Chrome, Firefox et Safari. Vous ne pouvez déboguer le code JavaScript que dans les navigateurs qui prennent en charge les cartes sources, comme Chrome.

ampouleConseil

Dans la mesure du possible, au lieu de déboguer le code de production, déboguez le code à l'aide d'un serveur de développement tel que celui fourni par [webdev](#).

Quel que soit le navigateur que vous utilisez, vous devez activer la pause au moins sur les exceptions non détectées, et peut-être sur toutes les exceptions. Pour les frameworks tels que `dart:async` ceux qui encapsulent le code utilisateur dans try-catch, nous recommandons de mettre en pause toutes les exceptions.

Chrome

#

Pour déboguer dans Chrome :

1. Ouvrez la fenêtre Outils de développement, comme décrit dans la [documentation Chrome DevTools](#).
2. Activez les cartes sources, comme décrit dans la vidéo [SourceMaps dans Chrome](#).
3. Activez le débogage, soit sur toutes les exceptions, soit uniquement sur les exceptions non détectées, comme décrit dans [Comment définir des points d'arrêt](#).
4. Rechargez votre application.

Bord

#

Pour déboguer dans Edge :

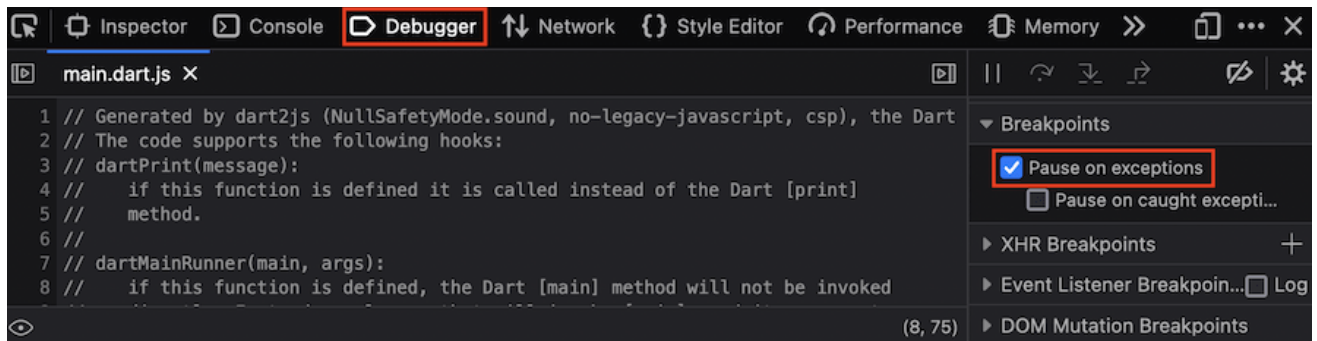
1. Mettez à jour vers la dernière version d'Edge.
2. Charger **les outils de développement** ([F12](#)).
3. Rechargez l'application. L'onglet **débogueur** affiche les fichiers mappés à la source.
4. Le comportement des exceptions peut être contrôlé via **Ctrl+Maj+E** ; la valeur par défaut est **Arrêt sur les exceptions non gérées** .

Firefox

#

Pour déboguer dans Firefox :

1. Ouvrez la fenêtre **Outils de développement Web** , comme décrit dans la [documentation des outils de développement Firefox](#) .
2. Activez **Pause sur les exceptions** , comme indiqué dans la figure suivante :



3. Rechargez l'application. L'onglet **Débogueur** affiche les fichiers mappés à la source.

Safari

#

Pour déboguer dans Safari :

1. Activez le menu **Développer** , comme décrit dans le [didacticiel Safari Web Inspector](#).
2. Activez les pauses, soit sur toutes les exceptions, soit uniquement sur les exceptions non détectées. Voir [Ajouter un point d'arrêt JavaScript](#) dans [l'aide du développeur Safari](#).
3. Rechargez votre application.

Ressources

#

Pour en savoir plus, consultez les éléments suivants :

- Documentation pour [votre IDE](#)
- [Documentation de Dart DevTools](#)
- [Documentation de l'outil webdev](#)
- [Documentation du package webdev](#)

Conclusion

Dans cette vidéo, nous avons exploré les **concepts clés** de la programmation en **Dart**, un langage de programmation polyvalent développé par Google. Voici un récapitulatif des principaux points abordés:

- **Variables et types de données:** Les variables sont utilisées pour stocker et manipuler des données. Les types de données de base comprennent les entiers, les décimaux, les chaînes de caractères et les booléens.
- **Structures de contrôle:** Les structures conditionnelles (if, else, else if) et les boucles (for, while) permettent de prendre des décisions et d'itérer sur des données pour gérer le flux d'exécution du programme.
- **Fonctions:** Les fonctions sont des blocs de code réutilisables qui effectuent des tâches spécifiques. Vous pouvez définir des fonctions avec des paramètres et des valeurs de renvoi.
- **Collections de données:** Les listes, les ensembles et les cartes sont utilisés pour stocker et manipuler des données de différentes manières, facilitant la gestion des informations dans vos programmes.
- **Classes et objets:** La programmation orientée objet en Dart repose sur les classes et les objets. Les classes définissent les propriétés et les méthodes, et les objets sont des instances de ces classes.
- **Gestion des erreurs:** Les blocs try-catch sont utilisés pour capturer et gérer les exceptions, permettant de gérer les erreurs de manière contrôlée.
- **Programmation asynchrone:** La programmation asynchrone est utilisée pour gérer les opérations qui prennent du temps à s'exécuter. Les futures, les promesses, et les opérateurs async/await sont utilisés pour gérer ces opérations de manière efficace.

Approfondir ses connaissances en Dart

La programmation est une compétence qui s'améliore avec la pratique constante. Continuez à écrire du code, à résoudre des problèmes et à explorer de **nouveaux concepts en Dart**. Explorez des projets plus complexes, participez à des communautés de développeurs et consultez la **documentation officielle** de Dart pour approfondir vos connaissances.

La maîtrise de Dart vous ouvrira des portes vers la **création d'applications mobiles avec Flutter**, le développement côté serveur, les applications web et bien plus encore. Continuez à construire, à apprendre et à grandir en tant que développeur Dart pour réaliser des projets passionnants et innovants. **Bonne programmation !**