Programmation asynchrone en Dart

Objectis

- Comprendre les bases de la programmation asynchrone
- Maîtriser les fonctionnalités asynchrones de Dart
- Gérer les opérations asynchrones
- Utiliser les bibliothèques asynchrones

1. ## Présentation

    - La programmation asynchrone est un style de programmation utilisé pour améliorer les performances d'un programme grâce à une utilisation efficace de son environnement d'exécution et des ressources système.
    - En Dart, la programmation asynchrone est prise en charge par le système de futures et de streams. Les futures représentent des valeurs qui peuvent ne pas être disponibles immédiatement, tandis que les streams sont des flux de valeurs qui peuvent être émises de manière asynchrone au fil du temps.

2. ## 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.
    - Dans un programme synchrone, les opérations sont effectuées les unes après les autres. Si une opération prend trop de temps (par exemple, une requête réseau), l'ensemble du programme est bloqué ou gelé jusqu'à ce que l'opération soit terminée. Cela peut entraîner une mauvaise expérience utilisateur, en particulier dans les applications d'interface utilisateur où la réactivité est cruciale.
    - La programmation asynchrone résout ce problème en permettant au programme de continuer à effectuer d'autres opérations en attendant la fin de l'opération de longue durée. Dans Dart, la programmation asynchrone est facilitée par l'utilisation de « Future », « async » et « wait ».
    - Les opérations asynchrones dans Dart peuvent être réalisées de deux manières :
        - Utiliser `wait` et `async`

apcpedagogie.com

- Utilisation de l'API `Future`.

3. `Future` en Dart

- En Dart, un "**Future**" représente une valeur potentielle ou une erreur qui sera disponible à un moment donné dans le future.
- C'est un moyen de traiter les opérations asynchrones de manière efficace. Lorsqu'une fonction retourne un "**Future**", elle promet essentiellement que la valeur sera calculée ou obtenue à un moment donné, mais pas nécessairement immédiatement.
- Cela permet au programme de continuer à s'exécuter de manière asynchrone pendant que l'opération se déroule en arrière-plan. Une fois que la valeur est prête, le "**Future**" est "complété" avec cette valeur, et le programme peut y accéder.
- Cela facilite la programmation des tâches qui prennent du temps, comme les requêtes réseau ou les accès aux fichiers, sans bloquer l'ensemble de l'application.
- Dans le contexte de la programmation asynchrone en Dart, un "Future" peut se trouver dans l'un des trois états suivants :

1. **`Future` Non complété**

   - Le "**Future**" non complété attend que l'activité asynchrone de la fonction se termine ou qu'une erreur survienne. Par exemple :

```
import 'dart:async';

void main() {
  Future<int> future = getNumberAsync();

  // Le futur n'est pas encore complété, le programme continue à
s'exécuter
  print("Début du programme");

  future.then((value) {
    // Lorsque le futur est complété avec une valeur, cette fonction est
appelée
    print("La valeur obtenue est : $value");
  }).catchError((error) {
    // En cas d'erreur, cette fonction est appelée
    print("Une erreur s'est produite : $error");
  });

  // Le programme continue à s'exécuter avant que le futur ne soit
```

```
complété
  print("Fin du programme");
}

Future<int> getNumberAsync() {
  return Future.delayed(Duration(seconds: 2), () {
    // Une simulation d'activité asynchrone qui prend 2 secondes
    // Ici, nous simulons la réussite en renvoyant un nombre
    return 42;
  });
}
```

2. **`Future` Complété avec une valeur**

   - Le "**Future**" complété avec une valeur est prêt et contient une valeur. Par exemple :

```
import 'dart:async';

void main() {
  Future<int> futureWithValue = Future.value(42);

  futureWithValue.then((value) {
    // Lorsque le futur est complété avec une valeur, cette fonction est
appelée
    print("La valeur obtenue est : $value");
  });
}
```

3. **`Future` Complété avec une erreur**

   - Le "**Future**" complété avec des données d'erreur indique qu'une erreur s'est produite lors de l'exécution de l'activité asynchrone. Par exemple :

```
import 'dart:async';

void main() {
  Future<int> futureWithError = Future.error("Une erreur s'est
produite");

  futureWithError.catchError((error) {
```

```
    // En cas d'erreur, cette fonction est appelée
    print("Une erreur s'est produite : $error");
  });
}
```

4. ## La fonction `Future.any()`

   - La fonction **Future.any()** renvoie le résultat du premier futur dans la liste des futurs fournis qui se complète. Voici une explication de l'exemple fourni :

```
Future<int> slowInt() async {
  await Future.delayed(const Duration(seconds: 2));
  return 2;
}

Future<String> delayedString() async {
  await Future.delayed(const Duration(seconds: 4));
  throw Exception('Time has passed');
}

Future<int> fastInt() async {
  await Future.delayed(const Duration(seconds: 1));
  return 3;
}

void main() async {
  final result = await Future.any([slowInt(), delayedString(), fastInt()]);
  // Le futur de fastInt se complète en premier, les autres sont ignorés.
  print('Résultat: $result'); // 3
}
```

   - Dans cet exemple :
     - La fonction **slowInt()** simule une opération asynchrone qui prend 2 secondes et renvoie finalement la valeur 2.
     - La fonction **delayedString()** simule une opération asynchrone qui prend 4 secondes et finalement lance une exception.
     - La fonction **fastInt()** simule une opération asynchrone qui prend 1 seconde et renvoie finalement la valeur 3.
     - Dans la fonction **main()**, nous utilisons Future.any() pour attendre que l'un des futurs se complète en premier. Comme **fastInt()** est le plus rapide, son futur se complète en premier. Les autres futurs ne sont pas pris en compte. Donc, le

apcpedagogie.com

résultat affiché est 3.

5. ## La fonction `Future.delayed()`

   - La fonction **Future.delayed()** crée un futur qui exécute son calcul après un délai spécifié. Voici une explication de l'exemple fourni :

```
void main() async {
  Future.delayed(Duration(seconds: 2), () {
    print('Ce message s'affichera après 2 secondes');
  });
}
```

   - Dans cet exemple :
     - La fonction **main()** est déclarée comme **asynchrone (async)** pour pouvoir utiliser l'attente (await) plus tard si nécessaire.
     - La fonction **Future.delayed()** crée un futur qui attend 2 secondes avant d'exécuter son calcul.
     - Le calcul est spécifié comme une fonction anonyme qui imprime un message après le délai de 2 secondes.
     - Comme la **fonction main()** ne contient pas d'attente explicite (await), le programme peut se terminer avant que le futur créé par **Future.delayed()** ne soit complet. Cela signifie que le programme peut se terminer avant que le message soit imprimé.
   - Pour garantir que le message est imprimé avant la fin du programme, vous devriez soit ajouter une attente explicite à la fin de la **fonction main()** à l'aide de await, soit utiliser .then() pour gérer le futur et attendre son achèvement. Par exemple :

```
void main() async {
  await Future.delayed(Duration(seconds: 2), () {
    print('Ce message s'affichera après 2 secondes');
  });
  print('Fin du programme');
}
```

   - Dans cet exemple, en ajoutant **await**, le programme attend que le futur se complète avant de continuer, garantissant ainsi que le message est imprimé avant la fin du programme.

apcpedagogie.com

6. ## Async/await

   - **Async/await** est une syntaxe utilisée dans de nombreux langages de programmation pour rendre le code **asynchrone** plus propre et plus lisible. Voici comment les mots-clés **async** et **await** sont utilisés ensemble :

```
Future<String> getLanguage() async {
  // La fonction est marquée comme asynchrone avec le mot-clé async
  await Future.delayed(Duration(seconds: 2));
  // L'appel coûteux est préfixé par le mot-clé await
  return 'Dart';
}

void main() async {
  print('Début du programme');
  // Utilisation de await pour attendre que la fonction getLanguage() se termine
  String language = await getLanguage();
  print('Le langage est : $language');
  print('Fin du programme');
}
```

   - Dans cet exemple :
     - La fonction **getLanguage()** est marquée comme asynchrone avec le mot-clé async, ce qui signifie qu'elle sera exécutée de manière asynchrone.
     - À l'intérieur de **getLanguage()**, l'appel coûteux à Future.delayed() est préfixé par le mot-clé await, ce qui indique au programme de suspendre l'exécution de cette fonction jusqu'à ce que le délai soit écoulé.
     - La fonction **main()** est également marquée comme asynchrone avec le mot-clé async, car elle utilise await.
     - L'appel à **getLanguage()** dans **main()** est préfixé par await, ce qui signifie que le programme attend que **getLanguage()** soit terminé avant de continuer.
     - Les messages d'impression sont affichés dans l'ordre correct en raison de l'utilisation d'**await**, qui garantit que les opérations asynchrones sont exécutées dans le bon ordre.

Cours Dart

apcpedagogie.com

Source : https://apcpedagogie.com/programmation-asynchrone-en-dart · téléchargé le 07/11/2026 · © APC Pédagogie · apcpedagogie.com

apcpedagogie.com