Introduction à Dart

m2iformation.fr

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Introduction à Dart

Objectifs pédagogiques

- Comprendre la **syntaxe de base** de Dart.
- Identifier les **points communs et différences** avec Java et JavaScript.
- Maîtriser les **structures de contrôle**, **fonctions**, **classes**, et **null safety**.
- Se préparer à écrire du code Flutter.

Présentation rapide de Dart

- **Langage développé par Google**
- Créé en 2011, utilisé d'abord pour le web, aujourd'hui principalement pour **Flutter**
- Compilé :
  - **AOT (Ahead-of-Time)** pour le mobile (performance proche du natif)
  - **JIT (Just-In-Time)** pour le développement rapide (hot reload)
- Syntaxe inspirée de **Java, JavaScript, C#**

Compilation AOT (Ahead-Of-Time)

Le code source est **compilé avant l'exécution** du programme.

✅ **Avantages:**

- Démarrage plus rapide
- Performances natives (pas de surcoût d'analyse à l'exécution)
- Pas besoin de machine virtuelle à l'exécution

❌ **Inconvénients:**

- Compilation plus lente au moment du build
- Moins flexible pour du développement interactif

Compilation JIT (Just-In-Time)

Le code source est **compilé dynamiquement pendant l'exécution**, souvent **au moment de son utilisation**.

✅ **Avantages:**

- Démarrage rapide (pas besoin de compiler tout de suite)
- Permet le Hot Reload / Hot Restart
- Possibilité d’optimiser en fonction de l'exécution réelle

❌ **Inconvénients:**

- Overhead mémoire/CPU
- Moins performant sur mobile ou embarqué

Dart et ses deux modes de compilation

Mode	Compilation	Usage	Avantages
Debug / Dev	**JIT**	Développement Flutter	Hot reload, cycle rapide
Release / Prod	**AOT**	Application en production	Performance, taille, sécurité

```
Code Dart
   │
   ├── JIT (Dev) -> flutter run
   │     └── Dart VM → Compilation partielle à la volée
   │           (hot reload)
   │
   └── AOT (Prod) -> flutter build
         └── Compilation complète → Binaire ARM
               (exécutable rapide, optimisé)
```

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Environnement

Environnement

Pour pratiquer les bases du langage Dart, plusieurs IDE sont possibles:

1. **DartPad** :
   - **Description** : DartPad est un outil en ligne qui vous permet d'écrire, d'exécuter et de partager du code Dart directement dans votre navigateur.
   - **Avantages** : Pas besoin d'installer quoi que ce soit, idéal pour les débutants et pour des tests rapides.
   - **Lien** : DartPad

Environnement

2. **Visual Studio Code (VS Code)** :
   - **Description** : Un éditeur de code source léger mais puissant qui prend en charge Dart via des extensions.
   - **Extensions utiles** : Dart et Flutter extensions pour VS Code.
   - **Avantages** : Hautement personnalisable, bon support pour le débogage et les tests.
   - **Lien** : Visual Studio Code

Environnement

3. **IntelliJ IDEA** :
   - **Description** : Un IDE puissant qui offre un excellent support pour Dart, notamment via le plugin Flutter.
   - **Avantages** : Intégration complète avec les outils de développement Dart et Flutter, bon pour les projets plus grands.
   - **Lien** : IntelliJ IDEA

Environnement

4. **Android Studio** :
   - **Description** : Bien que principalement utilisé pour le développement Android, Android Studio offre également un bon support pour Dart via Flutter.
   - **Avantages** : Intégration facile avec Flutter pour le développement d'applications mobiles.
   - **Lien** : Android Studio

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Les bases du langage

Structure d’un programme Dart

La structure d'un programme Dart est assez simple, elle ne nécessite qu'une fonction `main()` :

```dart
void main() {
  print('Hello, Dart!');
}
```

- Fonction `main()` = point d’entrée
- `print()` = équivalent à `System.out.println()` ou `console.log()`

Variables et types

Typage statique mais optionnel

```
var name = 'Alice';      // inférence
String name = 'Alice';   // typé explicitement
final age = 30;          // constante (runtime)
const pi = 3.14;         // constante (compile-time)
```

- Comparaison avec d'autres langages:

JavaScript	Java	Dart
`let`, `const`	`String`, `int`	`var`, `final`, `const`

Typage statique souple

Chaque variable a un type connu au moment de la compilation.
Cela permet des vérifications plus précises **avant exécution**, ce qui réduit les erreurs.
Mais Dart peut aussi inférer ce type automatiquement grâce au mot-clé `var`.

```
var name = 'Alice';      // Dart infère que c'est une String
String name = 'Alice';   // même chose, mais explicitement typé
```

- Dart = typage **statique**, comme Java (Java10+ pour l'inférence, uniquement locale, pas en Dart)
- Dart ≠ typage **dynamique pur**, comme JavaScript

Inférence de type

Le compilateur devine le type à partir de la valeur affectée.

```
var age = 25;          // => int
var isActive = true; // => bool
```

⚠️ Une fois inféré, le type **ne peut plus changer** : Dart n'est pas faiblement typé comme JavaScript.

Variables immuables : `final` vs `const`

Mot-clé	Signification	Moment de fixation	Exemple
`final`	Valeur immuable (une fois assignée)	À **l'exécution** (runtime)	`final date = DateTime.now();`
`const`	Constante **connue à la compilation**	À **la compilation**	`const pi = 3.14;`

- `final` = une seule affectation possible, mais valeur calculée à l'exécution.
- `const` = valeur figée **au moment de la compilation**. Plus stricte.

Types de base

- `int`, `double`, `bool`, `String`, `List`, `Map`, `Set`
- Tous sont **objets** (pas de types primitifs comme en Java)

```
int x = 42;
double y = 3.14;
bool isOk = true;
String greeting = 'Hello';
List<int> scores = [10, 20, 30];
Map<String, int> ages = {'Alice': 30, 'Bob': 25};
```

Types de base en Dart – avec définitions

Type Dart	Définition
`int`	Nombre entier (positif ou négatif), sans décimale.
`double`	Nombre à virgule flottante (nombre réel).
`bool`	Booléen, représentant `true` ou `false`.
`String`	Chaîne de caractères (texte).
`List<T>`	Liste ordonnée de valeurs de type `T`, comme un tableau.
`Map<K, V>`	Dictionnaire associant une clé (`K`) à une valeur (`V`).
`Set<T>`	Collection non ordonnée d'éléments uniques de type `T`.
`dynamic`	Peut contenir n'importe quel type, vérifié à l'exécution.
`Object`	Classe mère de tous les objets ; toutes les valeurs Dart en héritent.
`Null`	Type ne contenant qu'une seule valeur : `null`.

Primitifs vs Objets : quelle différence ?

Java par exemple distingue **types primitifs** (`int`, `double`, `boolean`) et **objets** (`String`, `Integer`, `List`, etc.).
Les primitifs ne peuvent pas être `null`.
Les objets ont des méthodes, mais les types primitifs n'en ont pas directement.

- **En Dart: Tout est objet.**
  - Même une simple valeur `42` est un objet de type `int`.
    On peut utiliser les méthodes : `42.isEven`, `3.14.round()`, `"Hello".length`
  - L'uniformité rend la syntaxe plus cohérente (comme en JS).

List et Set

Caractéristique	List	Set
Ordre	Ordonnée	Non ordonnée
Doublons	Autorisés	Non autorisés
Accès par index	Oui	Non
Utilisation	Ordre et accès par index	Unicité et vérification rapide

Si vous avez besoin d'accéder aux éléments par index, vous pouvez convertir le Set en List :

```
Set<String> fruits = {'pomme', 'banane', 'orange'};
List<String> fruitsList = fruits.toList();
print(fruitsList[0]); // Affiche le premier élément
```

var vs dynamic en Dart

🟢 var = type implicite mais fixé

```
var name = 'Alice';
```

- Dart **devine le type** (`String` ici) grâce à l'inférence.
- Mais ensuite, **le type est fixé** et ne peut plus changer :

```
var name = 'Alice';
name = 'Bob';          // OK
name = 42;             // ❌ Erreur : int n'est pas un String
```

var vs dynamic en Dart

dynamic = type non fixé, vérifié à l'exécution

```
dynamic anything = 'Hello';
```

- Dart **n'impose aucun type à la compilation**.
- On peut changer la valeur pour n'importe quel type :

```
dynamic anything = 'Hello';
anything = 42;              // OK
anything = true;            // OK
```

On **perd l'autocomplétion**, et on n'a **pas de vérification de type à la compilation**, donc plus d'erreurs possibles à l'exécution.

🧪 Exemples comparés

```
void main() {
  var x = 10;         // x est un int
  // x = 'hello';     // ❌ Erreur

  dynamic y = 10;     // y peut changer de type
  y = 'hello';        // ✅ OK

  print(x.isEven);    // ✅ OK, x est un int
  print(y.isEven);    // ⚠️ Erreur possible à l'exécution si y est String
}
```

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Contrôle de flux:

Boucles, Conditions, Fonctions

Structures disponibles en Dart

Structure	Description rapide
`if / else`	Conditionnel classique. Accolades obligatoires.
`for`	Boucle indexée classique (`for (int i = 0; i < n; i++)`).
`for-in`	Parcourt une collection (`for (var x in list)`).
`while`	Boucle tant que la condition est vraie.
`do while`	S'exécute au moins une fois.
`forEach`	Méthode d'objet sur les listes (`list.forEach((x) => ...)`).
`switch / case`	Existe en Dart avec `=>`, `default`, `continue`.

If / else

```
if (age > 18) {
  print("Adulte");
} else {
  print("Mineur");
}
```

For, while, forEach

```
for (int i = 0; i < 5; i++) {
  print(i);
}

scores.forEach((s) => print(s));
```

Quelques précisions

1. Pas de conversions implicites vers booléen

En Dart, la condition **doit** être un booléen (`true` / `false`).
Pas de “truthy / falsy” comme en JS (`""`, `0`, `null`, etc.), ni de conversion automatique comme parfois en Java.

```
if ('') { ... }       // ❌ Erreur
if (0) { ... }        // ❌ Erreur
if (someBool) { ... } // ✅ OK
```

Quelques précisions

2. Boucle `for-in` plus utilisée que l'indexée

En Dart, on manipule souvent des collections (List, Set, Map), et on cherche à itérer sur les éléments eux-mêmes, pas forcément sur les indices.
Comme Dart est orienté objet et moderne, il encourage cette syntaxe plus lisible.

```
for (var fruit in fruits) {
  print(fruit);
}
```

Quelques précisions

3. forEach est une méthode, pas une structure

En Dart, `forEach` est une méthode sur les collections (List, Set, etc.), **comme en JS**, mais **pas une syntaxe spéciale** comme `for (x of y)`.

```
myList.forEach((item) {
  print(item);
});
```

Notez donc la structure pointée sous forme de méthode.

Le Switch (structure classique)

- Dart permet des **expressions fléchées** (`=>`)
- Les `case` **peuvent continuer vers un autre cas** avec `continue`
- Dart **n'a pas besoin de** `break` à chaque case si on utilise `=>`
- En Dart, il n'y a pas de chute automatique d'un case à l'autre comme dans certains autres langages (par exemple, C ou Java).
- Le bloc **default** est optionnel

⚠️ Dart ne supporte pas les **cas multiples** (`case 1, 2:`), mais on peut chaîner avec `continue`.

Le Switch (structure classique)

```
switch (animal) {
  case 'dog':
    print('It barks');
    break;
  case 'cat':
    print('It meows');
    break;
  default:
    print('Unknown animal');
}
```

Le Switch (structure classique)

Version condensée :

```
switch (value) {
  case 1: print('Un'); break;
  case 2: print('Deux'); break;
  default: print('Autre');
}
```

Le Switch (expression)

Depuis Dart 2.12, une nouvelle syntaxe pour les instructions switch a été introduite, permettant une approche plus moderne et concise.

```
var result = switch (variable) {
    valeur1 => 'Résultat 1',
    valeur2 => 'Résultat 2',
    _ => 'Résultat par défaut',
};
```

La nouvelle syntaxe permet d'utiliser switch comme une expression, ce qui signifie qu'elle peut retourner une valeur.

Caractéristiques de la nouvelle syntaxe

- **Expression:** Le switch peut maintenant être utilisé comme une expression qui retourne une valeur. Cela permet d'assigner directement le résultat à une variable.
- **Flèches:** Les cas sont définis avec des flèches =>, ce qui rend le code plus concis.
- **Cas par défaut:** Le cas par défaut est représenté par _, similaire à certains autres langages de programmation fonctionnels.
- **Pas de break:** Contrairement à la syntaxe traditionnelle, il n'est pas nécessaire d'utiliser break.

Exemple concret

```
void main() {
    String grade = 'A';

    var result = switch (grade) {
        'A' => 'Excellent!',
        'B' => 'Bien!',
        'C' => 'Moyen.',
        'D' => 'Passable.',
        'F' => 'Échec.',
        _ => 'Note invalide.',
    };

    print(result);
}
```

Les fonctions

La syntaxe des fonctions en Dart est très classique.

```
// Déclaration classique
int add(int a, int b) {
  return a + b;
}

// Appel
print(add(2, 3)); // 5
```

On indique le **type de retour** et les **types des paramètres**.

Inférence de type sur les fonctions

L'inférence de type **fonctionne** également sur **les fonctions**

```
var result = add(2, 3); // Dart infère que result est un int
```

On peut aussi omettre le type de retour si on utilise `var`, mais le typage reste possible/fort.

Comme en JS, **les fonctions fléchés (arrow)** sont possibles (`=>`), très utile pour les fonctions simples à une seule expression.

```
int multiply(int a, int b) => a * b;
```

Paramètres nommés, optionnels, par défaut

Paramètres nommés (comme les objets JS)

```
void greet({String name = 'inconnu'}) {
  print("Bonjour $name");
}

greet();                  // Bonjour inconnu
greet(name: 'Alice');  // Bonjour Alice
```

Paramètres positionnels optionnels

```dart
void log(String message, [String? level]) {
  print('$level: $message');
}

log('Erreur');             // null: Erreur
log('Erreur', 'SEVERE');   // SEVERE: Erreur
```

➡️ Spécificité Dart : très utile en Flutter, car on appelle souvent des fonctions avec des objets de configuration.

Fonction comme valeur (fonction = objet)

```dart
void sayHello() => print('Hello');

void execute(Function f) {
  f(); // exécution différée
}

execute(sayHello); // Hello
```

Dart permet de **passer des fonctions en paramètres**.

Fonctions anonymes (lambdas / closures)

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

numbers.forEach((n) {
  print(n * 2);
});
```

Fonctions récursives

```
int factorial(int n) {
  if (n <= 1) return 1;
  return n * factorial(n - 1);
}
```

TOPIOS
TRANSFORMATION DIGITALE INTELLIGENTE

Gestion moderne de `null`

La Null Safety

Pourquoi la Null Safety ?

Le problème historique : le "billion-dollar mistake"

Dans les années 1960, Tony Hoare, créateur du concept de null, a introduit cette valeur comme une solution temporaire pour désigner « l'absence de donnée ».

Ce choix s'est généralisé dans presque tous les langages modernes: C, Java, C#, JavaScript, etc.

Il a plus tard reconnu que c'était "sa plus grosse erreur", ayant causé des milliards de dollars de bugs et de pannes.

Exemples classiques d'erreurs liées à null

Java:

```
String nom = null;
System.out.println(nom.length()); // NullPointerException
```

JavaScript:

```
let nom = null;
console.log(nom.length); // TypeError: Cannot read property 'length' of null
```

Dans les langages compilés, **ces erreurs sont invisibles** à la compilation, donc **le programme plante à l'exécution.**

Les solutions proposées par Dart

Avec la Null Safety, Dart:

- oblige le développeur à préciser explicitement qu'une variable peut être null.
- empêche d'accéder à une méthode ou une propriété sans avoir d'abord validé la présence d'une valeur.
- fournit des opérateurs dédiés pour gérer proprement la nullabilité.

Types non-nullables

Pour éviter tout ces problèmes, on ne veut plus que null soit un état par défaut.

Donc, **en Dart, par défaut, une variable ne peut pas être null.**

```
String nom = "Alice"; // OK
nom = null;           // ❌ Erreur à la compilation
```

Le langage va nous fournir 5 opérateurs à connaître pour gérer la nullabilité.

Les opérateurs liés à la nullabilité

`?` - Déclaration nullable

Ajouté après un type, il signifie que la variable **peut contenir** `null`.

```
int? age = null;
```

- **À utiliser uniquement si `null` est une valeur attendue ou temporaire.**

Les opérateurs liés à la nullabilité

`?.` – Appel conditionnel

Permet d'accéder à une propriété ou méthode **seulement si l'objet n'est pas** `null`. Retourne `null` sinon, sans erreur.

```
String? nom = null;
print(nom?.length); // Résultat : null (pas d'exception)
```

- **Évite les erreurs à l'exécution.**

Les opérateurs liés à la nullabilité

`??` – Valeur par défaut si null

Renvoie la **valeur de gauche si non-nulle**, sinon la valeur de droite.

```
String? titre = null;
print(titre ?? "Titre inconnu"); // Affiche : "Titre inconnu"
```

- **Utile pour les affichages ou valeurs fallback.**

Les opérateurs liés à la nullabilité

`??=` – Affectation conditionnelle

Affecte une valeur à la variable **uniquement si elle est actuellement** `null`.

```
int? x;
x ??= 10;
print(x); // 10
```

- **Évite des initialisations manuelles répétitives.**

Les opérateurs liés à la nullabilité

`!` – Assertion de non-nullité

Force Dart à considérer que la variable **n'est pas** `null`, même si elle l'est potentiellement.

```
String? nom = "Alice";
print(nom!.length); // ✅ OK

String? nom2 = null;
print(nom2!.length); // ❌ Exception à l'exécution
```

⚠️ **À utiliser uniquement si on est sûr que la variable n'est pas null.**

Exemples pratiques

```
void main() {
  String? titre;
  print(titre?.toUpperCase() ?? "Aucun titre");

  String auteur = "Jules Verne";
  String affiche = "${titre ?? "Titre inconnu"} - $auteur";
  print(affiche);
}
```

Bonnes pratiques

✅ À faire	❌ À éviter
Utiliser des types non-nullables par défaut	Rendre toutes les variables nullable "par sécurité"
Utiliser `??` pour fournir des valeurs par défaut	Abuser du `!` (assertion risquée)
Réfléchir au sens métier de `null` (est-il utile ou non ?)	Laisser des `null` sans traitement en aval

Gestion des collections et autres fonctionnalités

Qu’est-ce qu’une collection ?

En programmation, une **collection** est une **structure de données** utilisée pour stocker et manipuler **plusieurs valeurs** de façon organisée.
Dart propose trois structures de collection principales :

Collection	Description courte	Comparable à…
`List`	Liste ordonnée d’éléments, indexés	Tableau (JavaScript), `ArrayList` (Java)
`Set`	Ensemble non ordonné, sans doublons	`Set` en JavaScript / Java
`Map`	Table de hachage : paires clé → valeur	`Object` ou `Map` JS, `HashMap` Java

`List` – Liste ordonnée

Une **liste** est une collection ordonnée d'éléments. Chaque élément a un **indice** (commençant à 0).

Syntaxe de base

```
var fruits = ['pomme', 'banane', 'kiwi'];
print(fruits[0]); // pomme
```

Liste typée

```
List<String> prenoms = ['Alice', 'Bob'];
```

Méthodes utiles

Méthode	Description
`add()`	Ajoute un élément
`remove()`	Supprime un élément
`length`	Nombre d'éléments
`contains(x)`	Vérifie la présence d'un élément
`forEach()`	Itération sur tous les éléments

Quand utiliser une `List` ?

- Vous avez besoin d'un **ordre précis**.
- Vous accédez aux éléments par **indice**.
- Vous acceptez les doublons.

`Set` – Ensemble non ordonné sans doublons

Un **Set** est une collection **non ordonnée** d'éléments **uniques**.

```
var nombres = <int>{1, 2, 3};
nombres.add(2); // ignoré : déjà présent
```

Syntaxe typée

```
Set<String> tags = {'flutter', 'dart'};
```

Méthodes utiles du Set

Méthode	Description
`add()`	Ajoute un élément
`remove()`	Supprime un élément
`contains()`	Vérifie la présence d'un élément

- Vous voulez **empêcher les doublons**.
- L'**ordre n'est pas important**.
- Vous faites des opérations d'ensembles : unions, intersections, etc.

```
var a = {1, 2, 3}, var b = {3, 4};
var intersection = a.intersection(b); // {3}
```

`Map` – Paires clé / valeur

Une `Map` associe chaque **clé** à une **valeur**. Comparable à un dictionnaire, une table de hachage ou un objet JS.

```
var capitales = {
  'France': 'Paris',
  'Espagne': 'Madrid'
};
```

Syntaxe typée

```
Map<String, int> notes = {
  'Alice': 15,
  'Bob': 12,
};
```

Méthodes utiles de map

Méthode	Description
`putIfAbsent()`	Ajoute une clé si elle n'existe pas
`remove()`	Supprime une paire
`containsKey()`	Vérifie si une clé est présente
`forEach()`	Parcourt toutes les paires clé/valeur

```
notes['Claire'] = 17; // Ajout
```

Quand utiliser une `Map` ?

- Vous devez accéder à des valeurs **par clé**.
- Vous modélisez des **paires** : noms/valeurs, ID/objets, etc.

Comparatif : bien choisir sa collection

Besoin principal	Type conseillé
Conserver un ordre	`List`
Éviter les doublons	`Set`
Associer des clés à des valeurs	`Map`
Accès par position (index)	`List`
Rechercher une valeur rapidement	`Set` ou `Map`

Exemples d’usage concrets

- **Liste : liste de tâches**

```
List<String> taches = ['coder', 'tester', 'livrer'];
```

- **Set : étiquettes uniques**

```
Set<String> tags = {'dart', 'flutter', 'dart'}; // 'dart' n'apparaît qu'une fois
```

- **Map : carnet d’adresses**

```
Map<String, String> contact = {
  'Alice': 'alice@email.com',
  'Bob': 'bob@email.com',
};
```

Collections avancées

Spread Operator (`...`, `...?`)

Définition : Le *spread operator* permet de décomposer une collection dans une autre.

- `...` insère une collection entière dans une autre.
- `...?` fait la même chose, mais sans erreur si la collection est `null`.

Pourquoi ? Cela permet d'écrire des structures dynamiques plus lisibles et concises.

```
List<String>? noms = ['Alice', 'Bob'];
var tousLesNoms = ['John', ...?noms, 'Emma'];
```

Collection If / For

Définition : Dart permet d'inclure des conditions (`if`) ou des boucles (`for`) directement dans les collections. On parle de *collection literals conditionnels et itératifs*.

Avantage : Générer dynamiquement des listes ou maps sans devoir passer par des instructions séparées.

```
var estAdmin = true;
var droits = [
  'Lecture',
  if (estAdmin) 'Écriture',
  if (!estAdmin) 'Restreint',
];
```

```
var puissances = [
  for (var i = 1; i <= 3; i++)
  i * i
]; // [1, 4, 9]
```

Enums

a) Qu'est-ce qu'une Enum ?

Une `enum` (abréviation de *enumeration*) est un **type spécial** utilisé pour représenter un nombre limité de **valeurs constantes nommées**.
Exemple :

```
enum Jour { lundi, mardi, mercredi }
```

b) Évolution avec Dart 2.17

Dart permet désormais aux `enum` :

- d'avoir des **propriétés** et **méthodes** associées,
- d'être **construites** via des **constructeurs constants**.

```
enum Niveau {
  debutant(1),
  intermediaire(2),
  avance(3);

  final int code;
  const Niveau(this.code);

  String description() => 'Niveau $code';
}
```

c) ⚠️ Terminologie importante

Terme	Définition
`const constructor`	Constructeur évalué à la compilation
`getter`	Méthode sans parenthèses utilisée pour accéder à une propriété calculée
`name`	Nom brut de la valeur enum
`index`	Position dans la déclaration de l'enum (0, 1, 2...)

Cascade Notation (`..`)

La *cascade notation* permet d'**enchaîner plusieurs opérations sur un même objet** sans le réécrire plusieurs fois. Cela utilise l'opérateur `..`.

Exemple simple :

```
var buffer = StringBuffer()
  ..write('Salut')
  ..write(' tout le monde');
```

Cela permet une **écriture fluide**, notamment :

- pour configurer un objet (comme une UI Flutter),
- pour éviter le code répétitif.

TOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

Asynchronisme

Pourquoi l’asynchronisme ?

En Dart, comme dans beaucoup d’autres langages modernes, **l’asynchronisme permet d’exécuter des tâches longues (I/O, requêtes HTTP, timers…) sans bloquer l’exécution principale du programme**.

Dans une application Flutter, tout tourne autour d'une boucle d’événements qui doit **rester réactive** : bloquer cette boucle (par ex. en attendant une requête HTTP) figerait l’interface utilisateur.

Le type `Future<T>`

Un `Future<T>` est un **objet représentant une valeur ou une erreur qui sera disponible plus tard**, à la suite d'une opération asynchrone.

Déclaration simple

```
Future<String> fetchUserName() {
  return Future.delayed(Duration(seconds: 2), () => 'Alice');
}
```

Ici, `fetchUserName()` retourne immédiatement un `Future`, qui sera complété 2 secondes plus tard avec `'Alice'`.

Consommer un Future avec then

La méthode **then** est utilisée pour enchaîner des opérations asynchrones. Elle prend une fonction de rappel (callback) qui sera exécutée une fois que le Future sera complété avec une valeur.

```
fetchUserName().then((value) {
  print('Bonjour $value');
});
```

❗ Inconvénient : les `then()` imbriqués peuvent devenir difficiles à lire (callback hell).

Le mot-clé `async` / `await`

Dart permet une écriture **linéaire** et plus lisible grâce à `async` et `await`.

```
Future<void> greetUser() async {
  String name = await fetchUserName();
  print('Bonjour $name');
}
```

- `async` marque la fonction comme asynchrone
- `await` **suspend temporairement** l'exécution de la fonction jusqu'à ce que le `Future` soit résolu

Gestion des erreurs

La gestion des erreurs est importante en asynchrone.

En effet, les opérations asynchrones peuvent échouer pour diverses raisons, comme des problèmes de réseau, des erreurs de serveur, des fichiers introuvables, etc.

```
Future<void> greetUser() async {
  try {
    String name = await fetchUserName();
    print('Bonjour $name');
  } catch (e) {
    print('Erreur : $e');
  }
}
```

Le type `Stream<T>`

Un `Stream<T>` représente **une suite de valeurs asynchrones** dans le temps (contrairement à `Future` qui en renvoie une seule).

Exemple de `Stream`

```
Stream<int> generateNumbers() async* {
  for (int i = 0; i < 5; i++) {
    await Future.delayed(Duration(seconds: 1));
    yield i;
  }
}
```

Création de Stream avec async et yield

L'utilisation d'une fonction `async*` combinée à `yield` permet de générer dynamiquement un flux.

```dart
Stream<String> logEvents() async* {
  yield '📥 Début';
  await Future.delayed(Duration(seconds: 1));
  yield '⚙️ En cours';
  await Future.delayed(Duration(seconds: 1));
  yield '✅ Terminé';
}
```

Consommer un Stream

L'utilisation de `await for` permet de consommer les valeurs d'un Stream de manière asynchrone.

```
void main() async {
  await for (String event in logEvents()) {
    print('Événement : $event');
  }
}
```

L'exécution attend chaque yield successif du Stream.

UTOPIOS®
TRANSFORMATION DIGITALE INTELLIGENTE

La POO dans Dart

Définition d'une classe

```
class Person {
  String name;
  int age;

  Person(this.name, this.age); // constructeur court

  void sayHello() {
    print('Bonjour, je suis $name');
  }
}
```

- Les champs ne sont pas `private` par défaut.
- Pas besoin de `new` pour instancier (depuis Dart 2).

Constructeurs

Constructeur classique

```
Person(this.name, this.age);
```

Constructeur nommé

```
Person.anonyme() : name = '???', age = 0;
```

Constructeur avec paramètres nommés

```
Person.named({required this.name, required this.age});
```

Utilisation :

```
var p = Person.named(name: 'Alice', age: 30);
```

Encapsulation : visibilité et getters/setters

- Les membres `private` commencent par `_` (underscore)
- C'est **par fichier**, pas par classe !

```
class Compte {
  double _solde = 0;

  double get solde => _solde;

  void deposer(double montant) {
    _solde += montant;
  }
}
```

Héritage

```dart
class Animal {
  void parler() => print("...");
}

class Chien extends Animal {
  @override
  void parler() => print("Wouf !");
}
```

✅ Comme en Java :

- `extends` = héritage simple
- `@override` est **optionnel** mais recommandé

Classes abstraites

```
abstract class Forme {
  double aire();
}

class Cercle extends Forme {
  double rayon;
  Cercle(this.rayon);

  @override
  double aire() => 3.14 * rayon * rayon;
}
```

Interfaces et mixins

En Dart, **toute classe peut être utilisée comme interface** :

```dart
class Imprimable {
  void imprimer();
}

class Document implements Imprimable {
  @override
  void imprimer() => print("Impression...");
}
```

`implements` = on **doit redéfinir tout**.

Mixins : réutilisation de comportements

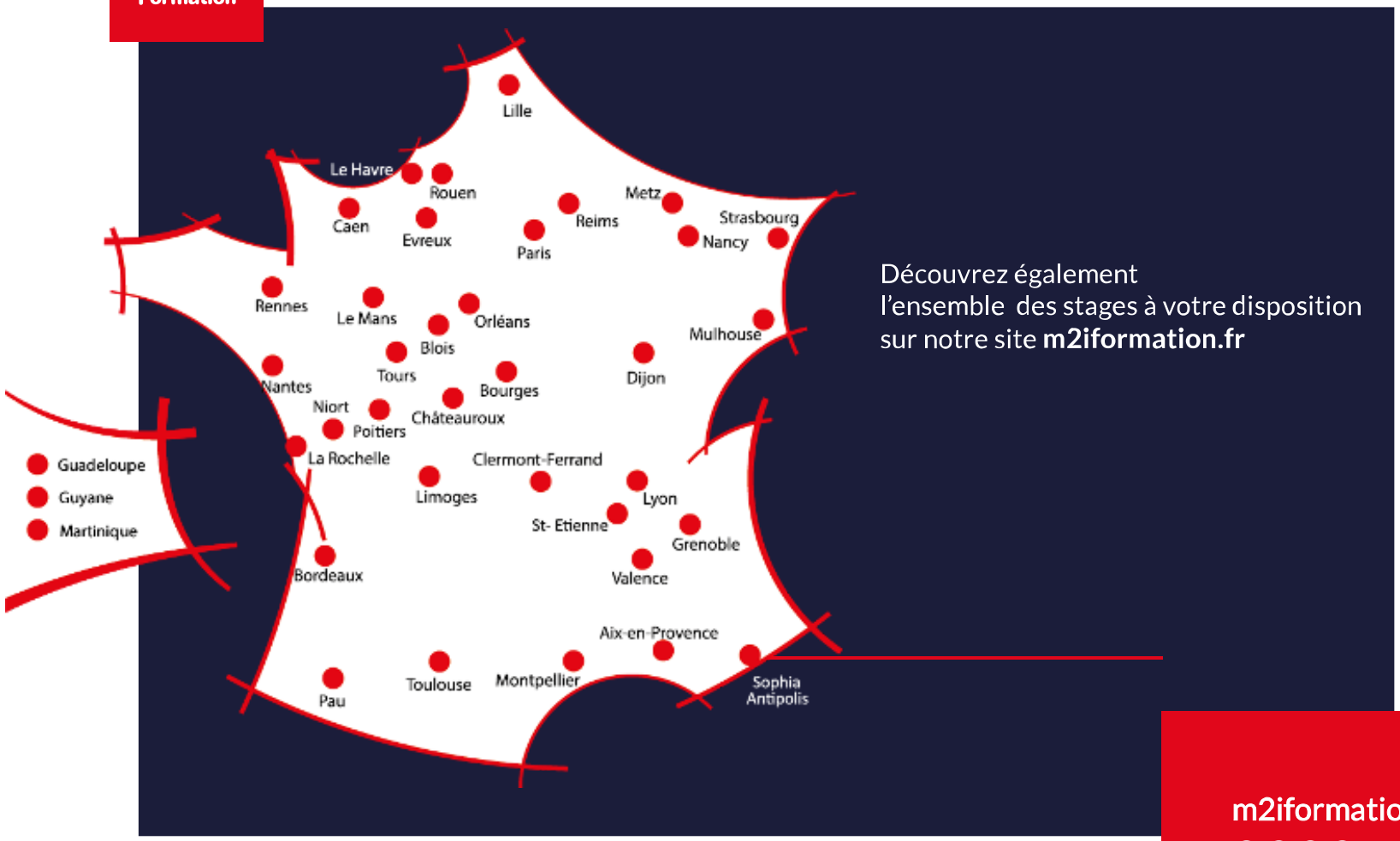
```
mixin Logger {
  void log(String msg) => print('[LOG] $msg');
}

class Service with Logger {
  void start() {
    log("Démarrage");
  }
}
```

Alternative légère à l'héritage pour partager du comportement.

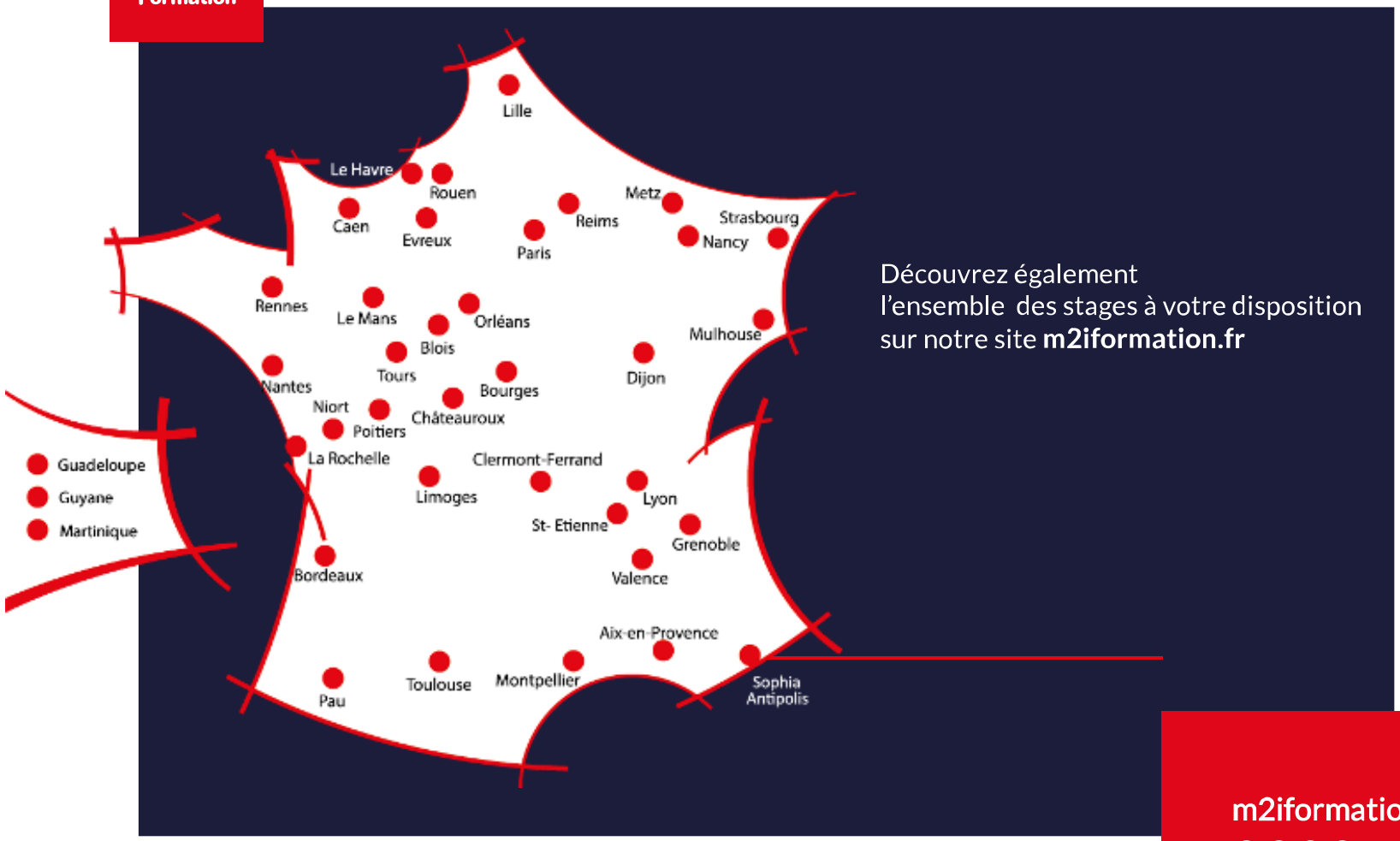
Merci pour votre attention

Des questions ?



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr



Découvrez également
l'ensemble des stages à votre disposition
sur notre site **m2iformation.fr**

m2iformation.fr