Flutter

Introduction à Flutter et Dart

Flutter ?

- **Flutter** est un framework pour le développement d'applications mobiles
  - multi-plateformes (code source unifié)
  - développé par Google
  - open-source
- https://flutter.dev

Multi-plateformes

- Android
- iOS (Mac nécessaire pour compiler)
- Desktop (Windows, macOS, Linux)
- Web
- Appareils embarqués

Dart ?

- **Dart** est le langage de programmation utilisé avec Flutter
  - développé par Google
  - famille des langages C (comme Java, C#, JavaScript…)
  - fortement typé
  - implémente la POO
  - implémente le *Null safety*
- Utilisé à la fois pour la logique métier et pour la description de l'UI
- Compile en code natif (ARM/x64) ou en JavaScript (Web)

DartPad

- Éditeur en ligne pour tester du code Dart
- https://dartpad.dev

```
void main() {
  print('Coucou !');
```

```
}
```

```
void main() {
  String name = 'Alice';
  print('Coucou ' + name + ' !'); // Concaténation
  print('Coucou $name !');         // Interpolation
}
```

`var` et `dynamic`

- `var` : **inférence de type**
  - permet de déclarer une variable sans préciser son type
  - le type est déterminé par Dart à la compilation
  - **cela reste un typage statique**
- `dynamic` : typage dynamique
  - le type est déterminé à l'exécution
  - **la variable peut changer de type**
- En Dart, le typage statique est beaucoup plus utilisé

```
void main() {
  var name = 'Alice';
  // name = 42; // Erreur de compilation (static)
  dynamic age = 42;
  print('Coucou $name, tu as $age ans !');
  age = 'quarante-deux'; // OK car dynamic
  print('Coucou $name, tu as $age ans !');
}
```

Déclaration de constantes

- `final` et `const` permettent de déclarer une constante
  - doit être initialisée à la déclaration
  - ne peut pas être modifiée
- Bonne pratique : toujours utiliser des constantes lorsque la valeur de la variable ne change pas
  - meilleures performances
  - pas de risque de modification non souhaitée (le compilateur nous en empêchera)
- Mais quelle est la différence entre `final` et `const` ?

`final` - constante à l'exécution

- Avec `final`, la valeur est déterminée **au moment de l'exécution**, quand le programme arrive sur la ligne en question
  - le type peut être omis, comme pour `var`
  - `final int age = 42;`
  - `final age = 42;` (idem)

`const` - constante à la compilation

- Avec `const`, la valeur est déterminée **au moment de la compilation**, quand l'exécutable est construit
  - **le type ne peut pas être omis**
  - la valeur doit être un **littéral** (valeur « en dur » : `42`, `'toto'`…)
  - `const int age = 42;`
  - `const age = 42; // erreur de compilation`

```
void main() {
  final name = 'Alice';  // constante
  // ici 'Alice' étant un littéral,
  // on pourrait aussi utiliser const en indiquant le type :
  // const String name = 'Alice';
  print('Coucou $name !');
  name = 'Bob'; // erreur de compilation (constante !)
}
```

final ou const ?

- Dans quels cas utiliser les constantes ?
  - toujours utiliser une constante (`final` ou `const`) lorsque la valeur ne change pas
- Et quel mot-clé utiliser alors ?
  - toujours privilégier `const` lorsque la *valeur est connue à la compilation*
  - utiliser `final` lorsque la valeur *ne peut être déterminée qu'à l'exécution*

Listes

- Dart supporte les listes génériques
- Syntaxe : `List<Type> variable = [valeur1, valeur2, ⋯];`

```
void main() {
  List<String> names = ['Alice', 'Bob', 'Eve'];
```

```
  names.add('Dave'); // 4 éléments dans la liste
  names.remove('Dave');
  names.removeAt(1); // supprime Bob (indices commencent à 0)
  print(names);      // [Alice, Eve]
  // Parcours de la liste
  for (int i = 0; i < names.length; i++) {
    print(names[i]);
  }
}
```

Parcours de liste amélioré

```
void main() {
  List<String> names = ['Alice', 'Bob', 'Eve'];
  // Syntaxe for spécifique pour les listes
  for (String name in names) {
    print(name);
  }
}
```

Fonctions

- La syntaxe est similaire à celle de Java
  - `TypeDeRetour nomFonction(Type1 parametre1, Type2 parametre2) { ⋯ }`

```
void main() {

  List<String> names = ['Alice', 'Bob', 'Eve'];
  if (containsName(names, 'Bob')) {
    print('Bob est dans la liste');
  } else {
    print('Bob n\'est pas dans la liste');
  }

  bool containsName(List<String> names, String name) {
    return names.contains(name);
  }
}
```

Arrow Functions

- Syntaxe plus concise pour les fonctions à une seule expression
- Utilisation d'une *fat arrow* ⇒, suppression du `return` et des accolades

```
void main() {

  List<String> names = ['Alice', 'Bob', 'Eve'];
  if (containsName(names, 'Bob')) {
    print('Bob est dans la liste');
  } else {
    print('Bob n\'est pas dans la liste');
  }

  // Équivalent à la fonction précédente
  bool containsName(List<String> names, String name) => names.contains(name);
}
```

Classes et Objets

- Dart est un langage orienté objet
- Syntaxe similaire à Java
  - `class NomClasse { ⋯ }`
  - particularité : `new` n'est pas obligatoire pour instancier un objet (en Flutter son usage est même *deprecated*)

```
void main() {
  IceCream vanilla = IceCream();
  vanilla.flavor = 'Vanilla';
  vanilla.isFruit = true;
  print(vanilla.flavor);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;
}
```

Constructeurs

- La déclaration des constructeurs est similaire à Java
  - Dart supporte également les constructeurs par défaut (exemple précédent)
- Syntaxe : `NomClasse(param1, param2) { ⋯ }`

```
void main() {
  IceCream chocolate = IceCream('Chocolate', false);
  print(chocolate.flavor);
}
```

```
class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(String flavor, bool isFruit) {
    this.flavor = flavor;
    this.isFruit = isFruit;
  }
}
```

Syntaxe raccourcie

- Dart propose une syntaxe raccourcie pour les constructeurs
  - en spécifiant le mot-clé `this` dans les paramètres du constructeur, il devient inutile de les assigner dans le corps du constructeur

```
void main() {
  IceCream vanilla = IceCream('Vanille', true);
  print(vanilla.flavor);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(this.flavor, this.isFruit);
}
```

Constructeurs nommés

- Dart ne supporte pas la surcharge de constructeurs
  - ⇒ un seul constructeur a le droit de prendre le nom de la classe
- Pour contourner cela, Dart propose les constructeurs nommés
  - Syntaxe : `NomClasse.nomConstructeur(param1, param2) { ⋯ }`

```
void main() {
  IceCream vanilla = IceCream('Vanille', true);
  IceCream chocolate = IceCream.withFlavor('Chocolat');
  IceCream strawberry = IceCream.fruitWithFlavor('Fraise');
  print(vanilla);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;
```

```
  // Constructeur non nommé (un seul)
  IceCream(this.flavor, this.isFruit);
  // Constructeur nommé
  IceCream.withFlavor(this.flavor);
  // Autre constructeur nommé
  IceCream.fruitWithFlavor(this.flavor) {
    isFruit = true;
  }
}
```

Héritage

- Dart supporte l'héritage simple
- Syntaxe : `class NomClasse extends ClasseParente { ⋯ }`
  - le mot-clé `super` est ensuite utilisé pour appeler le constructeur de la classe parente

```
void main() {
  Cone cone = Cone('Chocolat', false, true);
}

class IceCream {
  String flavor = '';
  bool isFruit = false;

  IceCream(this.flavor, this.isFruit);
}

class Cone extends IceCream {
  bool isGlutenFree = false;

  Cone(String flavor, bool isFruit, this.isGlutenFree)
      : super(flavor, isFruit);
}
```

Dart et la POO en bref

- `new` n'est pas nécessaire pour instancier un objet
- Un seul constructeur non nommé maximum
  - syntaxe `this.` dans les paramètres pour initialiser automatiquement les champs
- Autres constructeurs doivent être nommés
  - syntaxe `NomClasse.nomConstructeur(param1, param2) { }`
  - le pattern de nommage `with⋯` est ici souvent utilisé
- Héritage simple avec le mot-clé `extends`

- mot-clé `super` pour désigner la classe parente

Null safety

- En POO, les variables `null` sont à éviter au maximum
- Dart supporte plusieurs constructions pour implémenter la *Null safety* (sécurité contre les valeurs `null`)
  - types *nullables*, opérateurs `"??"`, `"?."`, `"??="`, opérateur *Bang* `"!"`
- Le *Null safety* aide à éviter les erreurs de type `null`
  - protection à la compilation
  - et aussi à l'exécution

```
class IceCream {
  // erreurs de compilation : champs 'null' interdits par défaut
  String flavor;
  bool isFruit;
}
```

Null safety : types *nullables*

- Solutions :
  - donner un valeur par défaut, comme auparavant : `String flavor = '';`
  - utiliser le constructeur non nommé avec la syntaxe `this.` en paramètre
  - **utiliser un type *nullable*** : `String? flavor;`
- Un type *nullable* est un type dont les variables peuvent être `null`
  - en Dart, il faut donc spécifier explicitement qu'on autorise une variable à être `null`

```
class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Null safety : pas de conversion implicite

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  String flavor = iceCream.flavor; // erreur de compilation
}

class IceCream {
```

```
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null coalescing*

- Le *Null coalescing* est un opérateur qui permet de retourner une valeur par défaut si une valeur est `null`
- Syntaxe : `valeur1 ?? valeur2`
  - si `valeur1` est `null`, alors `valeur2` est retournée
  - sinon, `valeur1` est retournée

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  // La valeur 'Inconnue' sera utilisée si flavor est null
  String flavor = iceCream.flavor ?? 'Inconnue';
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null assertion* ou opérateur *Bang* !

- L'opérateur *Bang* `!` permet de forcer l'accès à une variable qui peut être `null`
  - on l'utilise seulement quand on est certain que la variable n'est pas `null`
- Syntaxe : `variable!`
  - si `variable` est `null`, une erreur est levée
  - sinon, la variable est retournée

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Vanilla';
  // Ici le dev sait que flavor n'est pas null
  // Il force donc l'accès
  String flavor = iceCream.flavor!;
}

class IceCream {
  String? flavor;
  bool? isFruit;
```

```
}
```

Accès à un champ *nullable*

```
main() {
  IceCream iceCream = IceCream();
  String? flavor = iceCream.flavor; // OK
  print(flavor.length); // Erreur de compilation
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur *Null-aware*

- L'opérateur *Null-aware* `?.` permet d'accéder à un champ sur un objet qui peut être `null`
- Syntaxe : `objet?.champ`
  - si `objet` est `null`, alors `null` est retourné
  - sinon, la valeur de `champ` est utilisée
- On peut aussi l'utiliser pour un appel de méthode : `objet?.methode()`

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  String? flavor = iceCream.flavor;
  // Utilisation de ?.
  // OK, mais attention : la valeur peut être nulle
  print(flavor?.length);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur !.

- Même idée que l'opérateur *Bang* pour accéder à un champ *nullable*
  - ⇒ si vous êtes sûr que le champ n'est pas `null`, vous pouvez utiliser `!.` pour avoir une erreur si c'est le cas
- Syntaxe : `objet!.champ`

- si `objet` est `null`, une erreur est levée
- sinon, la valeur de `champ` est utilisée

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  String? flavor = iceCream.flavor;
  // Utilisation de !.
  // OK, mais si la valeur est nulle => erreur d'exécution
  print(flavor!.length);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Opérateur d'affectation *Null coalescing*

- L'opérateur d'affectation *Null coalescing* `??=` permet d'assigner une valeur par défaut à une variable si elle est `null`
- Syntaxe : `variable ??= valeur`
  - si `variable` est `null`, alors `valeur` est assignée à `variable`
  - sinon, `variable` reste inchangée

```
main() {
  IceCream iceCream = IceCream();
  iceCream.flavor = 'Chocolat';
  // Si flavor est null, on lui affecte 'Vanilla'
  iceCream.flavor ??= 'Vanilla';
  print(iceCream.flavor);
}

class IceCream {
  String? flavor;
  bool? isFruit;
}
```

Combinaison des opérateurs

- Tous ces opérateurs sont souvent combinés pour garantir le *Null safety* tout en gardant une syntaxe concise :
  - `objet?.champ ?? valeurParDefaut`
  - `objet?.champ!.methode() ?? valeurParDefaut`

- `objet?.champ ??= valeurParDefaut`