BUT2 RA Développement mobile - cours 2 :

Vous pouvez copier coller les encadrés de code de ce PDF dans un DartPad (`https://dartpad.dev/`) pour en voir le fonctionnement et le modifier comme vous le souhaitez

Sommaire :

    - Interface et classe abstraite
    - Lancer des exceptions
    - Catcher des exceptions
    - Héritage : notions avancées
- **Les éléments statiques**
    - Variables statiques
    - Fonctions statiques
- **Les enums**
    - Enums simples
    - Enums avec constructeurs

Créer une classe

Parametres non nommés, obligatoires, dans l'ordre

```dart
class Animal {
  final String id;
  final String name;
  final bool isExtinct;

  Animal( // <== constructeur sans accolades {}
    this.id,
    this.name,
    this.isExtinct,
  );
}

void main() {
  final ourson = Animal('1s23D', 'Knut', true); // <= création de l'instance
  print(ourson); // Output : Instance of 'Animal'
}
```

Parametres nommés, obligatoires, sans ordre

```dart
class Creature {
  final String id; // <== null impossible car ce n'est pas `String?`
  final String name;
  final bool isEvil;

  Creature({  // <== constructeur avec accolades {}
    required this.id, // <== REQUIRED
    required this.name,
    required this.isEvil,
  });
}

void main() {
  final fangorn = Creature(name :'Fangorn', id : 'aH145', isEvil : false);
  print(fangorn); // Output : Instance of 'Creature'
}
```

Parametres nommés, facultatif, sans ordre

- `id` est facultatif grace au point d'interrogation de `String?`. La valeur peut etre `null`.
- `name` est facultatif grace à la valeur par défaut dans le constructeur.
- `isPoisonous` est facultatif grace au point d'interrogation de `bool?`. Il ne sera null que s'il est passé en parametre

```dart
class Plant {
  final String? id;
  final String name;
  final bool? isPoisonous;

  Plant({  // <== constructeur avec accolades {}
    this.id,
    this.name = 'default name', // paramètre par défault
    this.isPoisonous = false, // paramètre par défault
  });
}

void main() => print(Plant(isPoisonous: null, id : '55oP4H'));
/// Output : Instance of 'Plant'
```

Getters et setters

```dart
class Etudiant {
  String? _name;
  String? _familyName;

  set name(String theName) => _name = theName; // <== SET

  String get name { // <== GET
    return _name ?? 'Pas de nom';
  }

  String get fullName => "$name $_familyName"; // <== GET
}

void main() {
  final etudiant = Etudiant();
  print(etudiant.name); // Output : 'Pas de nom'
  etudiant.name = 'Fernando';
  print(etudiant.name); // Output : 'Fernando'
}
```

L'héritage (avec extends)

Hériter d'une classe

- Mots clés : `extends`, `super`

```dart
class Animal {
  final bool isExtinct;
  Animal({required this.isExtinct});
}

class Cat extends Animal { // <== EXTENDS
  /// Ici, pas besoin d'écrire `isExtinct` et `type` car il sont dans `Animal`
  Cat({required super.isExtinct}); // <== SUPER à la place de this
}

void main() {
  final grympyChat = Cat(isExtinct : false);
  print("Extinct : ${grympyChat.isExtinct}"); // Output : "N'existe plus : false"
}
```

Redéfinir des parametres et fonctions : override

- Mots clés : `extends`, `super` et `@override`

```dart
void main() {
  final felixLeChat = Cat(name: 'Felix');

  print(felixLeChat.toString());
  /// Output : "name : Felix, type : Chat, isExtinct : false"

  print(felixLeChat.describe());
  /// Output : "Felix est un Chat. Cet animal existe encore sur terre"
}

class Animal { // <== Pas d'extends, c'est la classe parente
  final bool isExtinct;
  final String type;

  Animal({required this.isExtinct, required this.type});

  @override // <== OVERRIDE alors qu'on extend rien
  String toString() => "type : $type, isExtinct : $isExtinct";
```

```
  // Pas d'override
  String describe() =>
      isExtinct ? "Cet animal a disparu" : "Cet animal existe encore sur terre";
}

class Cat extends Animal { // <== EXTENDS
  final String name;
  /// Ici, pas besoin d'écrire `isExtinct` et `type` car il est dans `Animal`

  Cat({
    required this.name,
  }) : super(type: 'Chat', isExtinct: false); // <== SUPER

  @override // <== OVERRIDE
  String toString() => "name : $name, ${super.toString()}"; // <== SUPER

  @override // <== OVERRIDE
  String describe() => "$name est un $type. ${super.describe()}"; // <== SUPER
}
```

Explications de l'exemple précédent

- Mots clés : `extends`, `super` et `@override`
- `extends` : permet d'hériter d'une classe parente
- `super` : fait référence aux attributs et fonctions de la classe parente
- `@override`, deux cas :
    - dans la classe `Animal`, on `override` la fonction `.toString` alors qu'on n' `extend` rien. La raison est simplement que tout objet/classe en Dart/Flutter possède cette fonction.
    - dans la classe `Cat`, on `override` la fonction `desctiption` car on l'hérite de `Animal`. On peut soit la remplacer, soit la compléter avec une référence à `super.describe()`

Alternative à l'héritage multiple

- Une classe ne peut étendre qu'une seule autre classe. Mais on peut avoir le cas suivant :

```
class Vehicule {}

// Hérite de Vehicule
class Spaceship extends Vehicule {}

// Hérite de Vehicule ET Spaceship
class Rocket extends Spaceship {}
```

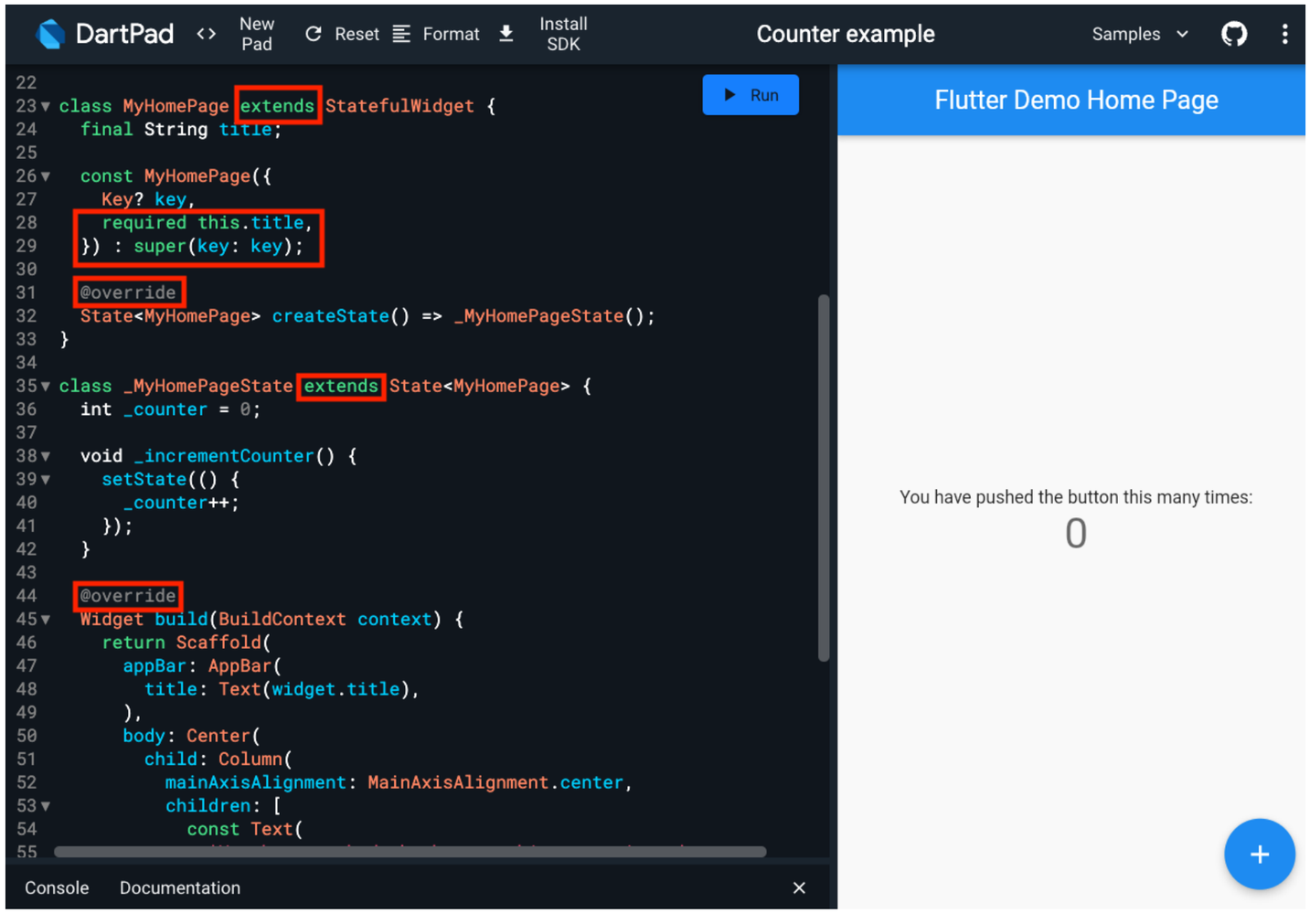
Alternative à l'héritage multiple (2)

- La fonction `afficher` attend un `Vehicule` en paramètre. On peut quand meme mettre les classes `Spaceship` et `Rocket` car elles héritent de `Vehicule`
- Avec les classes de la page précédente :

```dart
void main() {
  afficher(Vehicule()); // Output : Instance of 'Vehicule'

  afficher(Spaceship()); // Output : Instance of 'Spaceship'

  afficher(Rocket()); // Output : Instance of 'Rocket'
}

/// Cette fonction attend un Vehicule ou une classe qui en hérite
void afficher(Vehicule vehicule) {
  print(vehicule.toString());
}
```

Lien avec Flutter

- L'héritage vous sert à comprendre comment le rendu graphique est fait en Flutter.
- L'image suivante est un screenshot de l'appli de base de Flutter, créée automatiquement après la création d'un projet.



Les interfaces (avec implements)

Interface et classe abstraite

- Possibilité de définir des contrats que les classes doivent remplir
- Mots clé `abstract`, `implements`, `@override`

```
abstract class CalculateurAire {
  int aire(); // <= la fonction est déclarée mais sans body, càd sans {}
}

class Carre implements CalculateurAire { // <= IMPLEMENTS
  final int cote;
  Carre(this.cote);

  @override  // <= OVERRIDE
  int aire() {  // <= le body est définit avec {} et `return` quand on override
    return cote * cote;
  }
}
```

Lancer des exceptions

- Mots clé : `throw`, `try`, `catch`

```dart
void main() {
  try {
    throw Exception();
  } on Exception catch (e) {
    print(e); // Output : Exception
  }
}
```

- Pas besoin de créer la classe `Exception`, elle existe déjà en Dart et peut etre implémentée:

```dart
class ServerException implements Exception {}  // <= IMPLEMENTS

class ConnexionException implements Exception {  // <= IMPLEMENTS
  final String message = "Pas de connexion internet";
}
```

Catcher des exceptions

- Avec les classes `ServerException` et `ConnexionException` définit la page précédente :

```dart
void main() {
  try {
    throw ServerException(); // <== on lance une `ServerException`
  } on Exception catch (e) { // <== on la récupère dans `on Exception`
    print(e); // Output : Instance of 'ServerException'
  }

  try {
    throw ConnexionException();
  } on ConnexionException catch (e) {
    print(e.message); // Output : Pas de connexion internet
  } on Exception catch (e) {
    print(e); // <= l'exception n'est pas affichée ici car elle a été ratrapée
deux lignes avant dans `on ConnexionException`
  }
}
```

Héritage : notions avancées

- Il y a trois mots clés
    - `extends`
    - `implements`
    - `with`

L'article suivant explique la différence entre les trois. Le thème est un peu avancé, vous avez seulement besoin de savoir que cela existe pour l'instant. Vous pourrez lire cet article quand vous maitriserez l'héritage avec `extends`.

`https://medium.com/@manoelsrs/dart-extends-vs-implements-vs-with-b070f9637b36`

Les éléments statiques

Variables statiques

- Pas besoin d'instancer la classe `MyPaths` pour accéder à son contenu `static`

```dart
class MyPaths {
  static const String home = '/home'; // <== STATIC et CONST
  static const String camera = '/camera';
  static const String picturePreview = '${MyPaths.camera}/preview';
}

void main() {
  // Ci dessous, pas besoin d'écrire `final paths = MyPaths();` puis `paths.home`
  print(MyPaths.home);
  // Output : /home

  print(MyPaths.picturePreview);
  // Output : /camera/preview
}
```

Fonctions statiques

- Pas besoin d'instancer la classe `StringHelper` pour accéder à son contenu `static`

```dart
class StringHelper {
  static String loremIpsum() => 'Lorem ipsum dolor consectetur';

  static void cutText(String text, int limit) {
    final String resultat = text.substring(0, limit);
    print(resultat);
  }
}

void main() {
  print(StringHelper.loremIpsum());
  // Output : Lorem ipsum dolor consectetur

  StringHelper.cutText('Hello World', 5);
  // Output : Hello
}
```

Les enums

Enums simples

- Différent d'une `class`, pas besoin de l'instancier ni d'utiliser le mot `static`
- Permets de lister des éléments et d'accéder à cette liste avec `.values`

```
enum Status {
  nouveau,
  enCours,
  termine,
}

void main() {
  for (Status status in Status.values) // <== `.VALUES`
    print("$status / ${status.name}");

  // Parmis les outputs : Status.enCours / enCours
}
```

Enums avec constructeurs

- le constructeur doit etre constant, les valeurs ne seront jamais modifiée.

```
enum Status {
  nouveau("NOUVEAU"),
  enCours("EN COURS"),
  termine("TERMINÉ"); // <= point virgule

  const Status(this.label); // <= constructeur avec const

  final String label;

  String get description => "Le status est $label";
}

void main() {
  print(Status.enCours.description);
  // Output : Le status est EN COURS
}
```