Lien du dépôt : https://github.com/nterrien/cours_flutter_rendu_nicolasTERRIEN

* Intro (https://codelabs.developers.google.com/codelabs/from-java-to-dart/#0)

Les développement on été mis dans le dossier intro_to_dart_for_java_dev_code

Résumé des parties :
- Création de la classe Bycycle puis on supprime un des arguments du constructeur pour qu'il devienne readonly du coup on dois ajouter des méthodes pour incrémenter décrementer
- On a vu que les constructeurs en Dart pouvait prendre des arguments par défauts (comme les fonctions en python) ce qui rend certain argument optionnel. Cela rend un code plus court qu'en Java
- On a vu comment faire des factory. Cela peux se faire en dehors de toutes classe avec une fonction, ou alors avec le mot clef factory dans une classe). De plus on a vu qu'on pouvait faire plusieurs classe par fichier et que les classes abstraites existent.
- On a vu que toutes les classe sont des interfaces et que d'autre classe peuvent les implémenter comme en Java
- Les fonctions sont des objets et peuvent être donné en argument à d'autre fonctions. Les lambda expression existe en Dart.

* Stateless & stateful (https://flutter.dev/docs/development/ui/widgets-intro)

- Pour commencer une app il faut dans la fonction main, mettre runApp(Widget_principals)
- Une bonne pratique est d'utiliser MaterialApp comme Widget principale
- Pour détecter un clic cela se fait avec un widget GestureDetector, qui appelle un fonction onTap() lorsqu'il est toucher. Certain Widget (de Button par exemple) étendent ce widget et ont déjà une implémentation de onTap().
- Les widget Stateless n'ont pas d'état et ne changent pas alors que les widget Stateful utilisent des état (State) pour garder en mémoire certaines informations.
- Il y a une fonction dispose() qui est appelé lorsque que l'objet n'est plus utilisé, cette fonction peux être override.

* Layout (https://flutter.dev/docs/development/ui/layout/tutorial)

Cela corresponds au dossier building_layout sur le dépôt github
1 - On comprend comment est organisé une page en terme de lignes et de colonnes, la page exemple est organisée en Colonne qui contient des lignes qui contient des lignes ou des colonnes. On peux supposer qu'une page quelconque s'organise plus souvent en colonne à la base qu'en ligne (étant donné qu'un téléphone est plus souvent en format portrait que paysage).
2 - commit 9792c5d
3 - commit 5497d28
4 - commit e65503b
5 - commit 422ffc4
6 - commit ba697db. ListView supporte le scroll, il est préférable de l'utiliser plutôt qu'une simple colonne pour les cas ou l'écran est petit

* Interaction (https://flutter.dev/docs/development/ui/interactive)

On a appris à ajouter de l'interactivité dans l'application, cela correspond a changer les états de Widget lorsque que l'utilisateur clique dessus.
1 - Le widget que l'on veux créer n'impactent que lui même donc les changements d'état ne vont l'impacter que lui.
2 à 4 - commit 003e3cd -> Pour rentre les choses interactive on utilise la fonction onPressed du Widget IconButton pour appeler la fonction qui fait les changement que l'on souhaite.

Il y a des principes d'organisation de code pour savoir quel widget gère l'état d'un autre. Si c'est un effet purement esthétique cela est mieux géré par le widget lui même sinon si cela stocke des informations cela doit être géré par un widget parent. Il peux y avoir un gestion intermédiaire avec une partie de l'état géré par le widget lui même une autre par son parent.

Il y a d'autre Widget qui sont déjà existant et qui ont une interface utilisateur qui est faites pour être clair pour l'utilisateur.

* Navigation (https://flutter.dev/docs/development/ui/navigation)

- Pour animer un transition entre deux pages, il faut utiliser le widget Hero
- Pour passer d'une page (appelé route) à une autre on utilise Navigator.push(context,MaterialPageRoute()) pour rajouter une page par dessus celle qui est déjà affichée, et Navigator.pop() pour la retirer et repasser à la première page
- Pour ne pas avoir de code dupliqué il faut utiliser des route nommée. Par rapport à la méthode sans nom, il faut définir des routes dans un MaterialApp. Pour changer de page cela se fait avec Navigator.pushNamed() (qui utilise alors la nom donné dans la définition des routes au lieu d'une MaterialPageRoute) et pour revenir à la page d'avant avec Navigator.pop()
- Pour passer les données à une autre page lors de la navigation, il est possible d'utiliser Navigator.push et de donner les données aux Widget MaterialPageRoute. Il est également possible d'utiliser RouteSetting
- Pour passer les données à une page en utilisant une route nommée, il faut ajouter arguments: widget_avec_arguments. ou alors utiliser la fonction onGenerateRoute dans le MaterialApp pour extraire les données.
- Pour que la page sur laquelle on va retourne des données, il faut attendre lorsque qu'on lance la nouvelle page avec await devant Navigator.push(). Pour envoyer les données il faut mettre ces données dans le Navigator.pop(context, les_données)

* Changement d'état (https://flutter.dev/docs/development/data-and-backend/state-mgmt)

- Quand il y a un changement d'état l'interface utilisateur se redessine de zéro
- L'état de l'application peux être séparé en deux concepts
Ephemeral state et App State

- Ephemeral state sont des informations que l'on conserve que pour un widget et qui pourra assez vite être changé ou supprimé pendant l'utilisation de l'app
- App State sont les informations qui sont conservé entre plusieurs situation dans l'application, entre plusieurs page différents par exemple les préférence de l'utilisateur ne changent pas toutes seuls et on veux les garder
- Des données qui sont Ephemeral peuvent être App State si on se rend compte pendant le développement qu'elles sont utile à plusieurs widget
- Les States sont mis au dessus de leur Widget car lorsque que l'on change le Widget on le reconstruit mais le State on dois pas le reconstruire.
- ChangeNotifier est un widget qui peux notifier les Listeners (avec notifyListeners()) et donc permettre de mettre à jour l'interface utilisateurs de l'application
- Le ChangeNotifierProvider permet de donner accès à un ChangeNotifier aux Widgets qui descendent de lui.
- Consumer possède une méthode builder, qui est appelée à chaque fois que notifyListerners() est appelé. Il est préférable de mettre le Consumer le plus bas possible dans l'ordre des Widgets pour optimiser l'application sinon l'application reconstruit une trop grosse partie de l'interface pour un petit détail.
- Avec Provider.of il est possible de changer des élements sans avoir à les lire.

* 1er codelab (https://codelabs.developers.google.com/codelabs/first-flutter-app-pt1/index.html?index=..%2F..index#0 et https://codelabs.developers.google.com/codelabs/first-flutter-app-pt2/#3)

Ce codelab correspond au dossier write_your_first_flutter_app sur le dépôt github

- Commit 4470513 -> Création du projet avec le Hello world (Étape 3 du codelab)
- Commit e8473a2 -> On peux ajouter des dépendances dans le fichier pubspec.yaml dans la catégorie dependencies et en mettant import dans le fichier dart
- Commit 54be11e -> On ajoute un état de Mot aléatoire qui a chaque relance se recréer avec d'autre mots.
- Commit adf168a -> On stocke une liste de pair de mots qui sera vidé à chaque fois que l'on va descendre dans la ListView. Ce widget permet d'avoir une liste infiniment scrollable. Lorsque qu'il n'y a plus de mots dans suggestions on en régénère 10.

- Commit 751d789 -> Ajout d'icône de cœur a coté des mots qui ont la possibilité de se colorer en rouge si un mots est mis en favoris.
- Commit 476cbed -> Pour ajouter de l'interactivité, une fonction est appelé onTap() donc quand l'utilisateur appuie sur l'icône, cela ajoute ou retire les mots des favoris, avec ce qui a été fait dans l'étape précédente la couleur du cœur change en fonction.
- Commit 4e77db2 -> Pour naviguer on a ajouté une icône qui lorsqu'elle est cliqué appelle une fonction qui va push dans Navigator une Route qui est construite à ce moment la : Elle est constitué d'une liste des favoris.
- Commit 43dab42 -> On peux changer le thème d'une application en modifiant la classe ThemeData.

* 2nd codelab (https://codelabs.developers.google.com/codelabs/flutter/index.html?index=..%2F..index#0)

Ce codelab correspond au dossier building_beautiful_ui sur le dépôt github

- Commit b5b1e23 -> On y apprend que le Widget dans runApp est étendus sur tous l'écran et que home est la page par défaut.
- Commit 8597634 -> Il y a un widget TextField qui fait automatiquement ce qu'on attends d'un champs de texte éditable sur mobile, il ouvre le clavier virtuel. Pour avoir un Stateful widget comme on vois dans le cours à ce propos il faut étendre StatefulWidget et que le widget ait une State en parent. On apprend qu'il y a un IconThemeData qui permet de changer les couleurs des icônes dans l'app. On apprend dans cette étape à débugger en mettant des breakpoint comme on peux le faire sur avec d'autre éditeurs et d'autre langages.
- Commit ec16a5e -> On commence par implémenter une classe pour un message, puis on les mets dans un ListView qui permet d'avoir une liste scrollable
- Commit de7e63c -> On peut faire des animations, pour cela on utilise un Widget Animation. Le Widget AnimationController permet de définir des caractéristique sur l'animation. On peux mettre une durée pour les animations en donnant une Duration en argument. Parmi les animation il y a SizeTransition qui permet d'augmenter la taille de l'objet pendant sa transition et CurvedAnimation pour le faire glisser.
- Commit b02f349 -> On cherche ici a avoir un bouton d'envoi désactivé lorsque qu'il n'y a rien à envoyer, pour cela on ajoute un boolean qui indique s'il y a quelque chose dans le texte qui est écrit et le bouton ne fait rien si c'est le cas. Le Widget Expanded a permis de supprimer une erreur d'affichage lorsqu'il y avait trop de texte pour l'écran. Enfin on vois comment modifier le thème selon l'OS utiliser, avec defaultTargetPlatform pour connaître l'OS et l'attribut thème dans MaterialApp qui peut être modifié.