Flutter

Fiche 1 : layout & stateless widgets

Table des matières

1 Objectifs de la fiche

ID	Objectifs
F01	Création d'un stateless widget

2 Installation de l'environnement

Si vous utilisez votre propre machine et que ça n'est pas déjà fait, rendez-vous sur la page **Get started** de Flutter : https://flutter.dev/docs/get-started/install.

Suivez les instructions pour l'installation de Flutter selon votre plateforme (Windows/macOS/Linux) et pour un développement mobile/android, le développement web fonctionnera également.

Comme IDE, nous recommandons l'utilisation d'Android Studio. Suivez le lien dans les instructions pour l'installer. Vous devrez également y installer le plugin flutter.

Pour l'installation du Flutter SDK, choisissez l'onglet **Download and Install** plutôt que **Use VS Code to install**. À l'étape **Configure your target Android device**, installez le SDK Android 14.0, **API Level 34**, et choisissez un device récent comme le **Pixel 7 Pro**.

Lors de la vérification de l'installation avec **flutter doctor** sur Windows, il est probable que vous ayez ce « problème » identifié :

> [X] Visual Studio - develop for Windows

Nous vous proposons de ne pas résoudre ce « problème ». En effet, il n'est pas nécessaire de déployer nos applications Mobile en tant qu'application de bureau Windows.
Si vous avez ce souci identifié par **flutter doctor :**

> [!] Android Studio (version XX.X)
>
> X Unable to find bundled Java version.

La résolution est offerte sur stack overflow.

Pour les étudiants souhaitant travailler sur les machines de l'école, Android Studio et Flutter sont déjà installés. Il y a cependant certaines étapes à réaliser à chaque fois que vous vous connectez sur une machine pour la première fois pour que l'installation soit fonctionnelle :

- Ouvrez dans l'explorateur de fichier le dossier C:\Progs\flutter, ouvrez un Git Bash à cet emplacement et lancez la commande : **git config --global --add safe.directory '*'**
- Au premier lancement d'Android Studio, n'importez pas les paramètres d'une version précédente. Vous devez y installer le plugin flutter. En créant un projet flutter, Android Studio ne trouve pas automatiquement le chemin du **Flutter SDK path**. Vous devez y indiquer le chemin **C\:Progs\flutter**. Un SDK API level 34 et un premier AVD sont en revanche déjà installés, il n'est pas nécessaire de réaliser ces étapes.

3 Mise en place de votre GitHub repository pour les exercices

Nous vous demandons de travailler avec GitHub, via GitHub Classroom, pour versionner les projets que vous allez créer. Veuillez accéder à l'assignement du cours de Mobile sur GitHub Classroom : https://classroom.github.com/a/OY1BpkFi.

Loggez-vous au sein de GitHub via votre compte puis cliquez sur **Accept this assignment**. Rafraichissez votre page pour obtenir l'URL de votre web repo, par exemple `https://github.com/e-vinci/mobile-2023-exercises-e-baron`.

Cliquez sur l'URL de votre web repo. GitHub vous offre les instructions pour créer un nouveau repo local et le synchroniser avec votre GitHub repo. Créer un dossier **mobile** (ou donnez-lui le nom que vous voulez) sur votre machine à un endroit ne se trouvant pas sur un « cloud drive » comme OneDrive, Google Drive ou autre.

C'est ce dossier que vous allez transformer en repo local en le liant à votre GitHub repo. Entrez dans ce dossier à l'aide de Git Bash. Vous pouvez ensuite tapez les commandes offertes par votre GitHub repo sous « …or create a new repository on the command line ».

Pour chaque exercice, faites attention à créer un nouveau projet au sein d'un dossier dans ce repository (se trouvant dans votre dossier **mobile** ou autre) sur votre machine. Vous pourrez de cette façon utiliser l'interface de Android studio pour effectuer des commits et des push. Nous vous demanderons à plusieurs moments au sein des fiches du cours de faire des commits sur ce repository avec un tag **[Commit "message de commit"]**. Cela nous permettra de vérifier l'état de votre avancement dans la matière du cours, et votre implication.

Pour cette raison, nous vous demandons de bien vouloir suivre les messages de commits demandés. Vous pouvez faire d'autres commits intermédiaires si vous le souhaitez, mais nous nous attendons à retrouver ceux-là au minimum. Faites particulièrement attention aux commits correspondant à des objectifs du cours, dont le message commence par FXY.

4 Introduction au framework Flutter

Pour avoir une introduction à Flutter, veuillez :

1. Visionnez cette vidéo d'introduction (4 minutes) : How is flutter different for app devlopment.
2. Vous rendre sur la homepage du Framework : https://flutter.dev/ & lire les explications sur les objectifs (Fast, Productive, Flexible…) en cliquant notamment sur « Productive » et « Flexible ». N'hésitez pas à « jouer » quelques minutes (bouton « Try it in DartPad ») avec l'application exécutable dans votre browser.
3. Pour en savoir plus sur la philosophie de Design des UI dans Flutter, vous pouvez lire la (courte) page suivante : https://flutter.dev/docs/get-started/flutter-for/declarative

5 Introduction au langage Dart

Nous vous avons présenté notre Dart CheatSheet. Celle-ci se trouve sur Moodle.

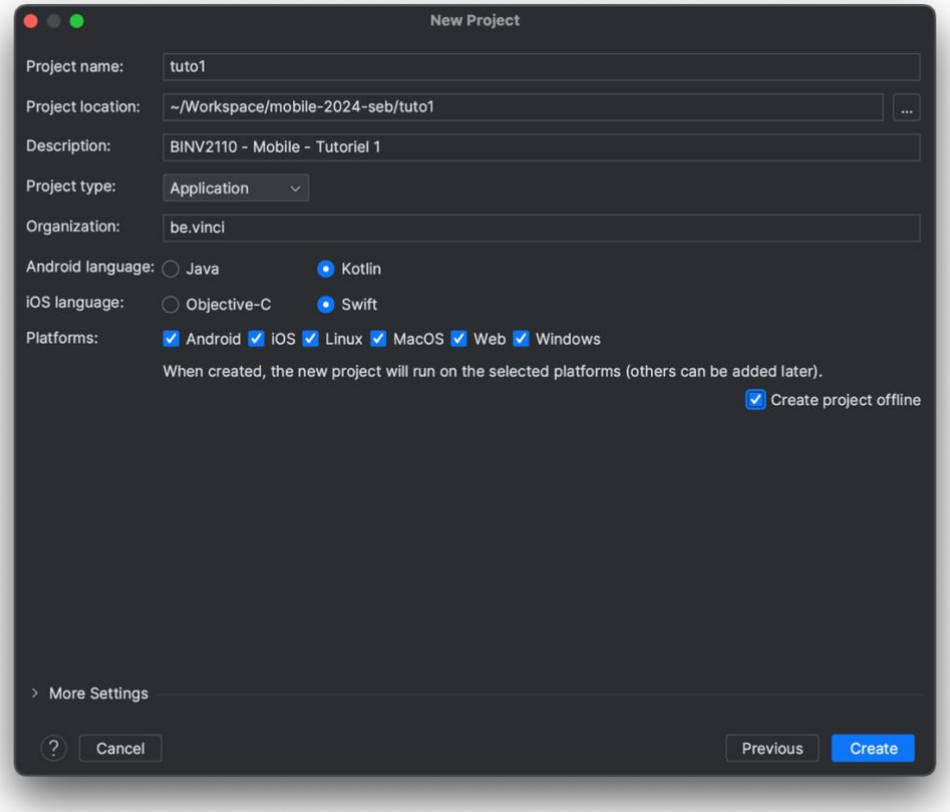
Toutefois si vous voulez avoir une vue plus détaillée du langage avant de commencer ou si durant la fiche vous aimeriez en savoir plus sur certaines constructions, rendez-vous sur le language-tour.

6 Concepts

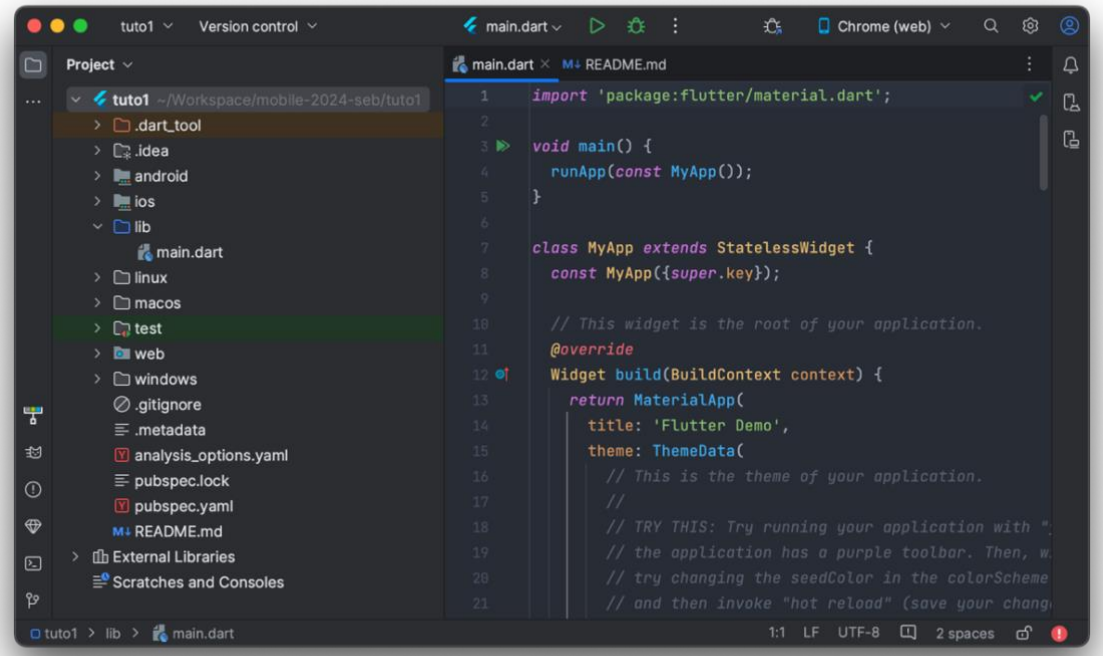
Dans ce premier tutoriel du cours, nous allons apprendre à créer des composants réutilisables pour composer notre affichage, ce que flutter appelle des **widgets**. Pour cela, nous allons créer ensemble une application présentant des produits en vente.

6.1 Introduction

Pour commencer le tutoriel, lancez **Android Studio** et créez un nouveau projet (**New Flutter Project…** et sélectionnez **Flutter**) nommé **tuto1** dans votre repository de cours. Vous pouvez laisser sélectionné toutes les plateformes avant de cliquer sur **Create**.



Le point d'entrée de l'application et le code de la starter app de flutter se trouve dans le fichier **lib/main.dart**. Dans un premier temps, nous ne devrions pas avoir à toucher aux dossiers spécifiques aux différentes plateformes de déploiement. Tout le code de l'app, identique à travers les différents déploiements, se fait dans le dossier **lib**.



Pour toutes les prochaines exécutions de vos programmes lors de la phase de développement, nous vous proposons de le faire pour la plateforme **Web**, via votre browser favori (Chrome par exemple). Ce type d'exécution consomme nettement moins de ressource de votre machine. Sélectionnez pour cela dans la barre supérieure la plateforme **Chrome (web)**. Autrement si vous avez un téléphone à disposition ou un ordinateur suffisamment puissant pour exécuter un **Virtual Device**, vous pouvez exécuter l'application sur celui-ci en changeant la plateforme dans ce même menu.

Lancez l'application pour vérifier que l'exécution de flutter se déroule correctement. Autrement, revenez à la section **Installation de l'environnement** pour vérifier que tout est correctement paramétré, ou demandez de l'aide à vos professeurs.

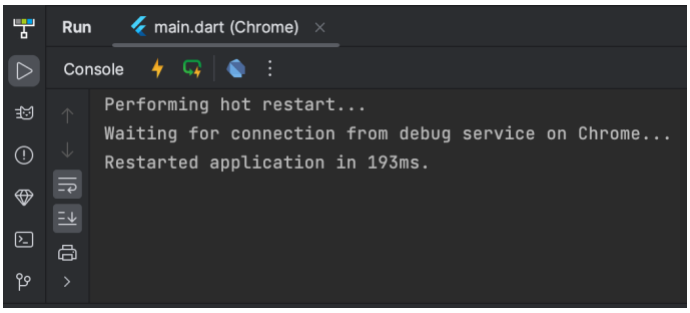
[commit avec message : T01.1 starter app]

Le code de cette starter app correspond à de la matière qui sera vue la semaine prochaine pour la fiche 2. Remplacez en attendant tout le contenu du fichier **lib/main.dart** par :

```
import 'package:flutter/material.dart';

void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Welcome to Flutter',
      theme: ThemeData(
```

```
        colorScheme: ColorScheme.fromSeed(seedColor: Colors.deepPurple),
        useMaterial3: true,
      ),
      home: Scaffold(
        appBar: AppBar(
          backgroundColor: Theme.of(context).colorScheme.inversePrimary,
          title: const Text('Welcome to Flutter'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

En enregistrant le projet ou en appuyant sur la touche en forme d'éclair dans l'onglet **Run**, l'application effectue un **Hot Reload**. L'application est réaffichée pour refléter les changements dans le code.



OBSERVATIONS & QUESTIONS

- Cet exemple crée une **MaterialApp**. Material est un système de conception visuelle qui est standard pour le Mobile et le Web. Flutter offre beaucoup de Material widgets.
- L'app étend **StatelessWidget**, c'est donc un widget. En Flutter, presque tout est un **Widget**, incluant l'alignement, le padding et le layout.
- Le **Scaffold** widget, de la librairie **Material**, fournit une app bar par défaut, un titre, et un propriété de type **body** qui contient l'arbre de widgets pour l'écran « **Home** ».
- Le job principal d'un widget est de fournir une méthode **build** qui décrit comment afficher le widget en terme de widgets "enfants".
- Le **body** consiste en un widget **Center** contenant un widget "enfant" de type **Text**. Le widget **Center** aligne son arbre de widgets au centre de l'écran.
- Le widget **MaterialApp** définit un thème, sur base d'un **ColorScheme** généré à partir d'une couleur principale de l'application. Ce thème est ensuite utilisé pour déterminer la couleur de fond de l'**AppBar**

Une fois le code analysé et exécuté : **[commit avec message : T01.2 app Flutter de base]**

6.2 Modélisation du layout

Les widgets les plus important en flutter sont **Row** et **Column**. Une **Row** permet d'organiser un ensemble de widgets enfants horizontalement, tandis qu'une **Column** permet de le faire verticalement. Il est également possible de les combiner, on peut par exemple faire une Column composé de Rows. D'autres widgets importants sont :

- **Text**, qui affiche un texte à l'écran
- **Center**, qui centre son widget enfant au sein de son widget parent
- **Padding**, qui ajoute de l'espace tout autour de son widget enfant
- **SizedBox**, qui contraint son widget enfant à une certaine taille. Utilisé sans enfant, il permet de créer un espace vide d'une certaine taille
- **Container**, qui permet via de nombreux paramètres de modifier l'affichage de son widget enfant, comme par exemple de changer sa couleur

En combinant ces différents widgets, il est possible de créer de nombreuses interfaces.

Remplacez le paramètre **body** du widget **Scaffold** avec les widgets suivants :

```
body: Center(
  child: Padding(
    padding: const EdgeInsets.all(8.0),
    child: SizedBox(
      width: 500,
      child: Column(
        children: [
          Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: [
              const Text(
                "iPhone 15 Pro Max",
                style: TextStyle(
                  fontWeight: FontWeight.bold,
                  fontSize: 16,
                ),
              ),
              Text(
                "1479 €",
                style: TextStyle(
                  color: Theme.of(context).colorScheme.primary,
                  fontSize: 16,
                ),
              ),
            ],
          ),
          const SizedBox(height: 8),
          const Text(
            "Lorem ipsum dolor sit amet, consectetur adipiscing elit. "
            "Aliquam et risus vel ipsum faucibus ultrices. "
            "Fusce nec leo nisi. Vestibulum vehicula, "
            "orci ac varius eleifend, ante erat efficitur tortor, "
            "quis tincidunt elit ex ut est.",
            textAlign: TextAlign.justify,
          )
        ],
      ),
    ),
  ),
),
```

OBSERVATIONS & QUESTIONS

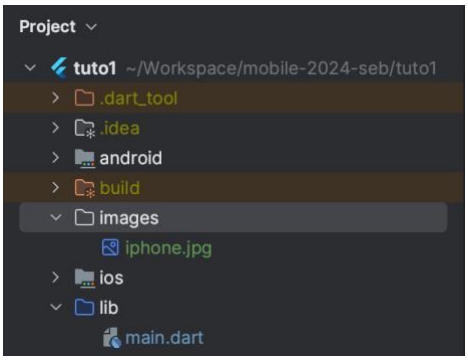
- Ce layout affiche une **Column**, avec comme enfants une **Row** et un **Text**, avec un espace de 8 pixels entre les deux au moyen d'une **SizedBox**.
- La **Row** affiche deux **Text** différents. Chacun possède un affichage différent grâce à l'utilisation d'un paramètre **style**. Ces deux **Text** sont organisés au sein de la **Row** au moyen du paramètre **mainAxisAlignemnt**. Un **MainAxisAlignment.spaceBetween** permet d'organiser les différents au sein d'une **Column** ou d'un **Row** en introduisant le plus d'espace possible entre eux. Dans ce cas, le premier **Text** de la **Row** est affiché le plus à gauche possible et le deuxième le plus à droite possible.
- L'ensemble de la **Column** est mis au sein d'un **SizedBox** pour limiter la largeur à 500 pixels, et au sein d'un **Padding** pour ajouter un espace de 8 pixels tout autour.

Réaffichez l'application avec ces changements, en effectuant un **Hot Reload**. Essayez de faire des modifications à l'affichage de ces widgets pour obtenir un layout qui vous convient.

[commit avec message : T01.3 un layout simple]

6.3 Ajout d'une image

Nous allons maintenant rajouter une image de description du produit. Récupérez depuis moodle l'image **iphone.jpg**, et ajoutez là dans un dossier **images** créé à la racine du projet.

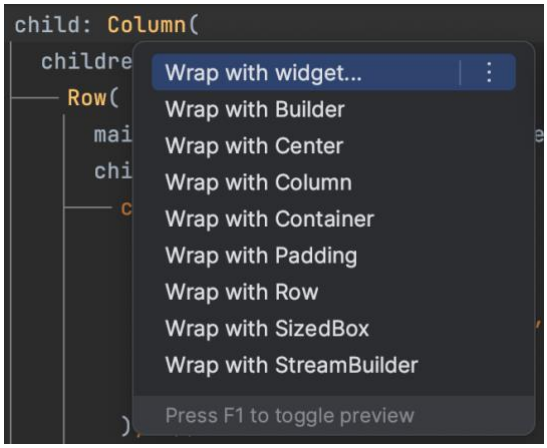


Pour intégrer une image à un projet flutter, il est nécessaire de la déclarer dans le fichier **pubspec.yaml**. Ce fichier est un fichier de configuration essentiel dans un projet Flutter. Il sert à déclarer les dépendances, les métadonnées du projet et les ressources telles que les images et les polices de caractères. Ajoutez la configuration suivante, dans la section **flutter** à la fin du fichier. Attention, les fichiers yaml utilisent l'indentation pour organiser les ressources. Il est donc très important de respecter l'indentation en modifiant ce fichier.

```yaml
flutter:

  # The following line ensures that the Material Icons font is
  # included with your application, so that you can use the icons in
  # the material Icons class.
  uses-material-design: true

  # Déclarez les images ici
  assets:
    - images/iphone.jpg
```

Après avoir modifié le fichier **pubspec.yaml**, il est nécessaire de cliquer sur le bouton **Pub get** dans le bandeau en haut de ce fichier pour charger les changements de configuration.

Il est possible de modifier facilement l'organisation des widgets dans Android Studio en sélectionnant un widget et en appuyant sur les touches **Alt+Enter**. Android Studio nous présente alors plusieurs possibilités, comme ajouter un widget parent, supprimer ce widget, etc. Modifiez la **Column** de votre layout actuel pour rajouter un widget parent **Row** en utilisant l'option **Wrap with Row**.



Ajoutez ensuite comme enfants de cette nouvelle **Row** les widgets suivants, à la suite de la **Column**.

```
const SizedBox(width: 16),
Image.asset('images/iphone.jpg', width: 150),
```

Si vous essayez de réafficher l'application à cette étape avec un **Hot Reload**, vous pourrez observer une erreur d'overflow dans votre application. Flutter n'arrive pas à deviner la taille que doit prendre les différents éléments au sein de la **Row** que nous venons de rajouter.

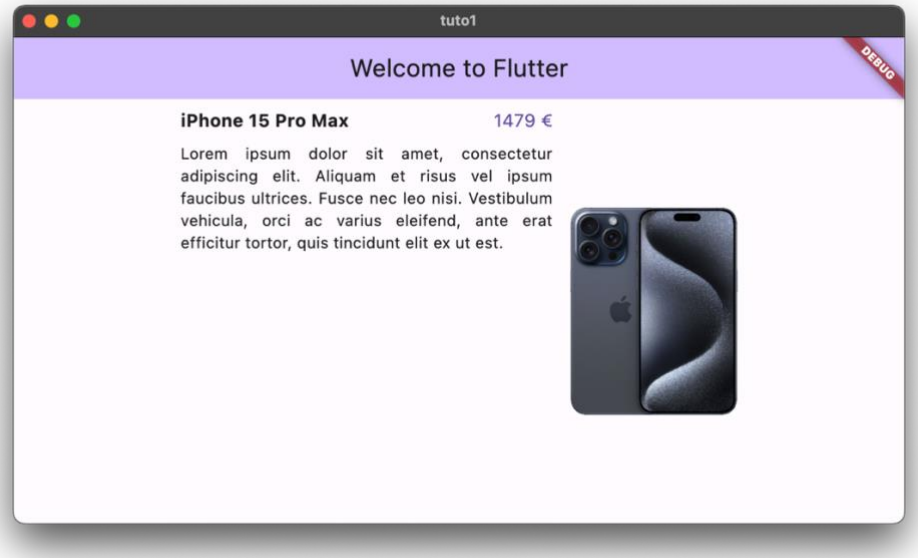
La **SizedBox** et l'**Image** ont une largeur fixée. Il est possible de faire en sorte que la **Column** prenne le reste de la largeur disponible (500 pixels au total via la **SizedBox**), en l'entourant d'un widget **Expanded**. Ce widget permet de faire qu'un enfant d'une **Row** ou une **Column** remplisse l'espace restant suivant les autres enfants. Utilisez l'option **Wrap with widget…** en sélectionnant la **Column** et écrivez ensuite **Expanded** comme widget.

Suite à tous ces changements, vous devriez obtenir le code suivant :

```
body: Center(
  child: Padding(
    padding: const EdgeInsets.all(8.0),
    child: SizedBox(
      width: 500,
      child: Row(
        children: [
          Expanded(
            child: Column(
              children: [
                Row(
                  mainAxisAlignment: MainAxisAlignment.spaceBetween,
                  children: [
                    const Text(
                      "iPhone 15 Pro Max",
                      style: TextStyle(
                        fontWeight: FontWeight.bold,
```

```
                        fontSize: 16,
                      ),
                    ),
                    Text(
                      "1479 €",
                      style: TextStyle(
                        color: Theme.of(context).colorScheme.primary,
                        fontSize: 16,
                      ),
                    ),
                  ],
                ),
                const SizedBox(height: 8),
                const Text(
                  "Lorem ipsum dolor sit amet, consectetur adipiscing elit.
"
                  "Aliquam et risus vel ipsum faucibus ultrices. "
                  "Fusce nec leo nisi. Vestibulum vehicula, "
                  "orci ac varius eleifend, ante erat efficitur tortor, "
                  "quis tincidunt elit ex ut est.",
                  textAlign: TextAlign.justify,
                ),
              ],
            ),
          ),
          const SizedBox(width: 16),
          Image.asset('images/iphone.jpg', width: 150),
        ],
      ),
    ),
  ),
),
```

Effectuez un **Hot Reload**. L'application devrait maintenant ressembler au screenshot suivant. Pour le moment, la **Row** prend tout l'espace vertical de l'application, ce qui fait que l'affichage est un peu étrange avec l'image qui est centrée. Ce problème sera réglé quand on affichera plusieurs produits.



[commit avec message : T01.4 ajout d'une image]

6.4 Création d'un widget réutilisable

Suite à ces derniers changements, le code de l'application est devenu assez long. On voudrait également pouvoir afficher plusieurs produits, ce qui ne serait pas simple de cette façon. Nous allons alors créer un widget réutilisable affichant notre produit.

Pour créer un widget, on peut utiliser le raccourci **stless** de Android Studio. Cela nous donne le code de base d'un widget, qu'il suffit alors de compléter. Dans ce cas cependant, nous avons déjà le code que nous voulons proposer comme widget. Il est possible de refactor ce code automatiquement avec Android Studio.

Faites un clic droit sur le widget **Padding**. Dans le menu **Refactor**, utilisez l'option **Extract Flutter Widget…** Donnez le nom **ProductWidget** pour le nouveau widget créé. Android Studio va automatiquement récupérer le code du **Padding** et tous ses widgets enfants, et l'inclure comme résultat de la méthode **build** du nouveau widget créé.

En Flutter, un stateless widget est simplement une classe qui étend **StatelessWidget**. Il y a également des **StatefulWidget**, que nous verrons dans la fiche 2. Chaque widget possède également une méthode **build** qui renvoie un widget décrivant l'affichage de ce widget, avec tous ses widgets enfants.

Automatiquement, Android Studio a créé votre widget **ProductWidget** dans le fichier **main.dart**. En Dart, il est possible de créer autant de classes et de méthodes que l'on souhaite dans un même fichier. On préférera quand même malgré ça créer des fichiers séparés pour nos différents widgets pour mieux structurer notre code. Créez un fichier **product_widget.dart** dans le dossier **lib** avec le menu **New** et l'option **Dart File**. Coupez la classe **ProductWidget** et collez-la dans ce nouveau fichier. Réglez les problèmes d'import dans les différents fichiers automatiquement avec Android Studio.

Après ces étapes, votre fichier **product_widget.dart** devrait ressembler au code suivant :

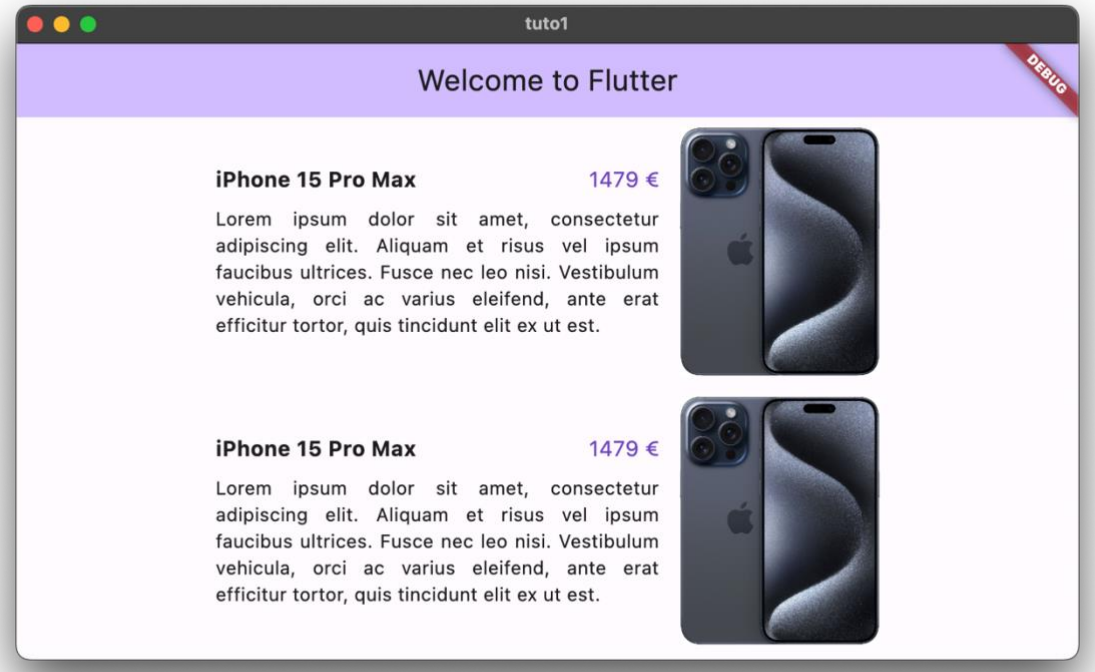
```dart
import 'package:flutter/material.dart';

class ProductWidget extends StatelessWidget {
  const ProductWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Padding(
      padding: const EdgeInsets.all(8.0),
      child: SizedBox(
        width: 500,
        child: Row(
          children: [
            Expanded(
              child: Column(
                children: [
                  Row(
                    mainAxisAlignment: MainAxisAlignment.spaceBetween,
                    children: [
                      const Text(
                        "iPhone 15 Pro Max",
                        style: TextStyle(
                          fontWeight: FontWeight.bold,
                          fontSize: 16,
                        ),
                      ),
                      Text(
                        "1479 €",
                        style: TextStyle(
```

```
                        color: Theme.of(context).colorScheme.primary,
                        fontSize: 16,
                      ),
                    ),
                  ],
                ),
                const SizedBox(height: 8),
                const Text(
                  "Lorem ipsum dolor sit amet, consectetur adipiscing
elit. "
                  "Aliquam et risus vel ipsum faucibus ultrices. "
                  "Fusce nec leo nisi. Vestibulum vehicula, "
                  "orci ac varius eleifend, ante erat efficitur tortor, "
                  "quis tincidunt elit ex ut est.",
                  textAlign: TextAlign.justify,
                ),
              ],
            ),
          ),
          const SizedBox(width: 16),
          Image.asset('images/iphone.jpg', width: 150),
        ],
      ),
    ),
  );
  }
}
```

Si vous réaffichez l'application à cette étape, l'affichage ne devrait pas avoir changé. Nous allons maintenant réutiliser le widget **ProductWidget** pour l'afficher deux fois dans l'application. Enveloppez le **ProductWidget** avec une **Column** dans le fichier **main.dart**. Ajoutez ensuite à cette **Column** un deuxième **ProductWidget**. Cela devrait donner le code suivant :

```
body: const Center(
  child: Column(
    children: [
      ProductWidget(),
      ProductWidget(),
    ],
  ),
),
```

En réaffichant l'application, le résultat devrait ressembler au screenshot suivant. Nous avons de cette façon créé notre propre widget réutilisable.



[commit avec message : T01.5 un widget réutilisable]

En créant cette **Column**, l'image n'est plus centrée au milieu de la page et est correctement alignée avec le texte. Cela vient du fait que la **Column** implique que les éléments à l'intérieur minimisent leur espace vertical. Sans cela, les widgets enfants d'un **Scaffold** occupent tout l'espace disponible. La **Row** a comme comportement par défaut de centrer ses éléments verticalement dans l'espace disponible.

Avant de mettre une **Column** autour de la **Row**, tout l'espace vertical de l'application est disponible et l'image s'y retrouve donc centré. Après l'ajout de cette **Column**, l'espace vertical de la **Row** est limité à son plus grand élément, c'est-à-dire l'image. C'est alors le texte à gauche de l'image qui se retrouve centré par rapport à elle au sein du **ProductWidget**.

6.5 Widget paramétrable

Nous allons maintenant modifier le widget **ProductWidget** pour qu'il puisse afficher tout type de produit. Commencez par rajouter les attributs suivant à la classe **ProductWidget** et modifiez son constructeur comme suit :

```
final String name;
final String description;
final int price;
final String imagePath;

const ProductWidget({
  super.key,
  required this.name,
  required this.description,
  required this.price,
  required this.imagePath,
});
```

Nous pouvons maintenant modifier l'affichage du **ProductWidget** pour utiliser ces paramètres au lieu des données précédemment hardcodées. Remplacez le code suivant :

```
Expanded(
  child: Column(
    children: [
      Row(
        mainAxisAlignment: MainAxisAlignment.spaceBetween,
        children: [
          Text(
            name,
            style: const TextStyle(
              fontWeight: FontWeight.bold,
              fontSize: 16,
            ),
          ),
          Text(
            "$price €",
            style: TextStyle(
              color: Theme.of(context).colorScheme.primary,
              fontSize: 16,
            ),
          ),
        ],
      ),
      const SizedBox(height: 8),
      Text(
        description,
        textAlign: TextAlign.justify,
      ),
    ],
  ),
),
const SizedBox(width: 16),
Image.asset(imagePath, width: 150),
```

Il ne nous reste plus qu'à modifier la création de nos **ProductWidget** dans le fichier **main.dart** pour préciser les produits affichés. Rajoutez le fichier **cg.jpg** dans le projet, comme vu dans la section 6.3 : ajoutez l'image dans le dossier **image**, modifiez le fichier **pubspec.yaml** pour déclarer l'asset, lancez le **pub get** pour valider la modification de la configuration du projet, et relancez l'application pour y intégrer le fichier. Intégrez ensuite le code suivant.

```
ProductWidget(
  name: "iPhone 15 Pro Max",
  price: 1479,
  description:
      "Lorem ipsum dolor sit amet, consectetur adipiscing elit. "
      "Aliquam et risus vel ipsum faucibus ultrices. "
      "Fusce nec leo nisi. Vestibulum vehicula, "
      "orci ac varius eleifend, ante erat efficitur tortor, "
      "quis tincidunt elit ex ut est.",
  imagePath: "images/iphone.jpg",
),
ProductWidget(
  name: "GeForce RTX 4080 SUPER",
  price: 1110,
  description:
      "Des jeux ultra-réalistes, ultra-fluides et ultra-immersifs, "
      "la carte graphique GeForce RTX 4080 SUPER met à votre "
```

```
      "disposition les technologies les plus avancées pour "
      "vous permettre de jouer dans les meilleures conditions, "
      "en très haute résolution ou en Réalité Virtuelle.",
  imagePath: "images/cg.jpg",
),
```

Après tous ces changements, votre application devrait ressembler à l'image suivante.

[commit avec message : T01.6 un widget paramétrisable]

OBSERVATIONS & QUESTIONS

- Tous les paramètres utilisés dans cette section sont des paramètres nommés. Il s'agit du style de paramètre le plus utilisé en Flutter. Essayez de modifier ces arguments en paramètres positionnels. Que devez-vous changer pour y arriver ?

6.6 Pour aller plus loin

Félicitations, vous avez créé vos premiers layouts en Flutter 🎉 ! Néanmoins, ça n'est pas évident de comprendre comment les largeurs, hauteurs, alignements… sont gérés. Les layouts en Flutter sont très différents des layouts en HTML / CSS. Mais plus vous allez pratiquer Flutter, plus cela va devenir naturel. Voici une règle très importante : **les contraintes vont vers le bas, les tailles vers le haut et le parent définit la position.**

Voici, de manière plus détaillée, comment le layout des widgets est déterminé :

- Un widget reçoit ses 4 **contraintes** de son **parent**. : minimum & maximum width, minimum & maximum height.
- Ensuite le widget va indiquer à ses **enfants** leurs **contraintes**, puis il demande à chaque enfant quelle **taille** il veut être.
- Ensuite le widget **positionne** ses enfants, **horizontalement** selon **l'axe x**, **verticalement** selon **l'axe y**, un par un.
- Finalement, le widget dit à son **parent** sa **taille** en respectant les contraintes reçues.

Voici les limitations :

- Un widget peut donc décider de sa taille seulement si cette taille respecte les contraintes reçues du parents !
- Un widget ne peut pas décider de sa position à l'écran (c'est le parent qui le fait) !
- Il faut prendre tout l'arbre des widgets pour déterminer la position d'un widget en particulier (c'est pas possible de le faire de manière isolée)
- La taille d'un enfant peut être ignorée si un parent n'a pas suffisamment d'info comment aligner un enfant qui aurait une taille différente.

Pour bien comprendre comment sont gérés les contraintes de taille et de position, vous pouvez jeter un œil aux exemples donnés sur : Understanding constraints.

Si vous souhaitez, vous pouvez également consulter les ressources suivantes pour en apprendre plus sur la création de widgets et de layout :

- How to create stateless widgets ? (vidéo de 7 minutes en anglais)
- Building user interfaces in Flutter (article et codelabs en anglais)
- Widget – State – Context - InheritedWidget (article en français)

Ne commencez pas encore les parties liées aux **StatefulWidget** dans ces différentes ressources, ce sera l'objectif de la fiche suivante !

7 Exercice

7.1 Introduction

Mais qui connait les prix Nobel ? Peut-être connaissez-vous le prix Nobel concernant la création de la dynamite… mais sinon, toutes ces sortes de « super héros » (scientifiques, artistiques, bienveillants…) sont bien souvent méconnus du public 😢.

Ils risquent donc de sombrer dans l'oubli…

Et c'est là que nous, créateurs d'apps, nous allons intervenir : remettons en lumière ces événements associés au prix Nobel.

Une API Open Source nous offre toutes les données nécessaires :
http://api.nobelprize.org/v1/prize.json

Veuillez créer un nouveau projet (New Flutter Project) nommé *ex1* dans votre repository de cours.

7.2 HomeScreen via un stateless widget

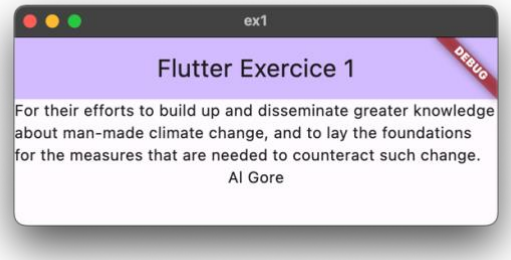
Nous souhaitons créer un **HomeScreen**, un stateless widget, qui affiche les données d'un lauréat d'un prix Nobel. Le **HomeScreen** doit contenir un **Scaffold** et un titre. Dans un premier temps, le code ne doit pas être trop structuré et toute l'UI se trouve dans le fichier **main.dart**.

Les données du lauréat doivent se trouver de cette **map** que vous ajouterez tout à la fin de votre fichier **main.dart** :

```
const laureate1 = {
  "id": "819",
  "firstname": "Al",
  "surname": "Gore",
  "motivation":
      "For their efforts to build up and disseminate greater knowledge
about man-made climate change, and to lay the foundations for the measures
that are needed to counteract such change.",
  "share": "2"
};
```

Veuillez-vous « nourrir » de cette map pour afficher la **motivation**, puis le **firstname** et **surname**.

Voilà à quoi pourrait ressembler votre application. **Ne passez pas trop de temps à peaufiner votre layout à cette étape-ci** !



⚡ **[commit avec message : F01.1 création du HomeScreen]** ⚡

7.3 Création d'un stateless widget paramétrable

Dans l'optique d'une future application qui pourra afficher de nombreux lauréats, vous devez créer un stateless widget qui doit :

- être paramétrable et réutilisable. Il doit donc se trouver dans une **librairie** ;
- indiquer le **firstname**, **surname** du lauréat ainsi que la **motivation** de lui avoir donné le prix Nobel ; ces informations doivent être données par des arguments **nommés**, ne pouvant pas être nuls :
    - **firstname** est optionnel, sa valeur par défaut est "" ;
    - **surname** est optionnel, sa valeur par défaut est "" ;
    - **motivation** est obligatoire.

Votre **HomeScreen** doit appeler votre nouveau widget présentant un lauréat.

Quand tout est fonctionnel comme au point précédent (fin du §7.2) :

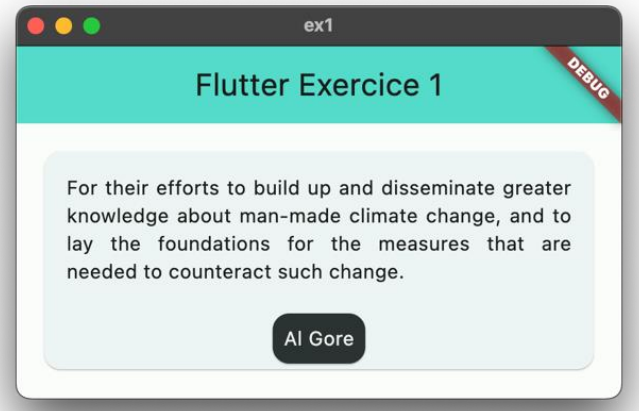
⚡ **[commit avec message : F01.2 création d'un widget paramétrable dans une librairie]** ⚡

7.4 Peaufinage du layout

Veuillez changer le layout de votre application. Vous devez :

- Jouer avec la couleur d'un des textes.
- Ajouter de l'espace entre le titre du **HomeScreen** et l'affichage du lauréat.

Voilà à quoi pourrait ressembler votre application.



Si vous essayez d'avoir un layout qui correspond à l'écran proposé, cela risque de vous prendre beaucoup de temps. N'hésitez pas à simplifier le layout, en allant au plus simple.

Besoin d'inspiration pour les layouts ? layout widgets et Components.

[commit avec message : F01.3 stateless widget peaufiné]