

Les bases de Flutter 2/2

Table des matières

I. Widgets de base	3
II. Exercice : Quiz	13
III. États	13
IV. Exercice : Quiz	28
V. Essentiel	28
VI. Auto-évaluation	29
A. Exercice :	29
B. Test	31
Solutions des exercices	32

I. Widgets de base

Contexte

Vous savez que « *tout est widget* ». Dans une application de base, le widget racine « *MyApp* » retourne un widget « *MaterialApp* » au moment de sa construction.

Ainsi, dans ce widget « *MaterialApp* », on spécifie le nom de l'application à travers la propriété « *title* ». Toujours dans ce widget, la propriété « *home* » permet de spécifier le point d'entrée de l'application.

Jusqu'à présent, on a utilisé des widgets « *StatelessWidget* ». Ces derniers n'ont pas d'état. Ils affichent juste de l'information. De plus, on sait construire un widget qui sera passé à la propriété « *home* » du widget « *MaterialApp* ». À l'intérieur de son constructeur, le widget retourne un widget « *Scaffold* ». Celui-ci met en place le squelette de l'application, dont la propriété « *body* » permet de construire l'interface utilisateur.

En premier lieu, pour construire le widget principal de notre application, on va utiliser d'autres widgets de base : container, Text, Image, Icon, Column, Row, Expanded. D'autre part, on sait comment intégrer dans le projet une image ou une police de caractères. Ainsi, l'application est personnalisable.

De plus, on va voir comment combiner des widgets à travers plusieurs fichiers. Jusqu'à présent, on avait vu des widgets sans interaction de l'utilisateur. Ici, on va aborder des widgets avec un état qui va changer selon les interactions de l'utilisateur. C'est le cas d'un bouton.

Ainsi, on abordera les « *widgets* » dont l'état va changer au fil de leur vie. On va découvrir un ensemble de widgets interactifs.

Au final, on pourra construire une application qui va permettre à l'utilisateur de saisir, par exemple, des données comme la fiche d'un produit.

Séparation des widgets en plusieurs fichiers

Dans un but de lisibilité du projet, il est préférable de découper son code dans différents fichiers à l'extension (.dart). On va créer un widget dans un fichier séparé et voir comment on l'appelle dans le main.dart.

On crée un fichier « *carte page.dart* » dans le répertoire lib. On crée un widget « *cartePage* ».

```
1 import 'package:flutter/material.dart';
2 class CartePage extends StatelessWidget {
3   @override
4   Widget build(BuildContext context) {
5     return(
6       Scaffold(
7         appBar: AppBar(
8           title: Text("Carte de visite"),
9         ),
10      body: Center(
11        child: Card(
12          child: Container(
13            height: 150,
14            width: 300,
15            color:Colors.grey
16          ),
17        ),
18      ),
19    ),
20  );
21 };
22 }
23 }
```

Dans notre fichier `main.dart`, on va importer le fichier « `carte_page.dart` ». On appellera notre widget `CartePage()`.

```

1 import 'package:flutter/material.dart';
2 import 'package:exercicefinal/carte_page.dart';
3
4 void main() {
5   runApp(MyApp());
6 }
7 class MyApp extends StatelessWidget {
8   @override
9   Widget build(BuildContext context) {
10    return MaterialApp(
11      title: 'Flutter Demo',
12      theme: ThemeData(
13        primarySwatch: Colors.blue,
14      ),
15      home: CartePage()
16    );
17  }
18 }
```

CircleAvatar

Le widget « *CircleAvatar* » est un cercle à l'intérieur duquel on peut mettre une couleur, une image ou un texte d'arrière-plan.

```

1 CircleAvatar(
2   foregroundImage: AssetImage("assets/images/beach.png") ,
3   radius: 30,
4 ),
```

Ici, on définit un cercle ayant pour propriété un radius de 30 et une image avec la propriété `foregroundImage`.

Dans la carte, on utilise dans le container un widget « *Row* » pour aligner horizontalement un widget « *CircleAvartar* » et un « *text* ».

```

1 Container(
2   child: Row(
3     children: [
4       Padding(
5         padding: const EdgeInsets.all(8.0),
6         child: CircleAvatar(
7           foregroundImage: AssetImage("assets/images/beach.png") ,
8           radius: 30,
9         ),
10      ),
11      Text("La Plage",style: TextStyle(
12        fontSize: 45,
13        color: Colors.white
14      ),
15    ),
16    ],
17  ),
18  height: 150,
19  width: 300,
20  color:Colors.grey
21 ),
```



Pour l'image, on aurait pu charger l'image d'arrière-plan avec le widget « *NetworkImage* ».

Divider

Le widget « *Divider* » a pour fonction de créer une ligne séparatrice horizontale. Son utilisation est courante dans le « *drawer* » (menu) pour séparer la liste des onglets.

Ses propriétés :

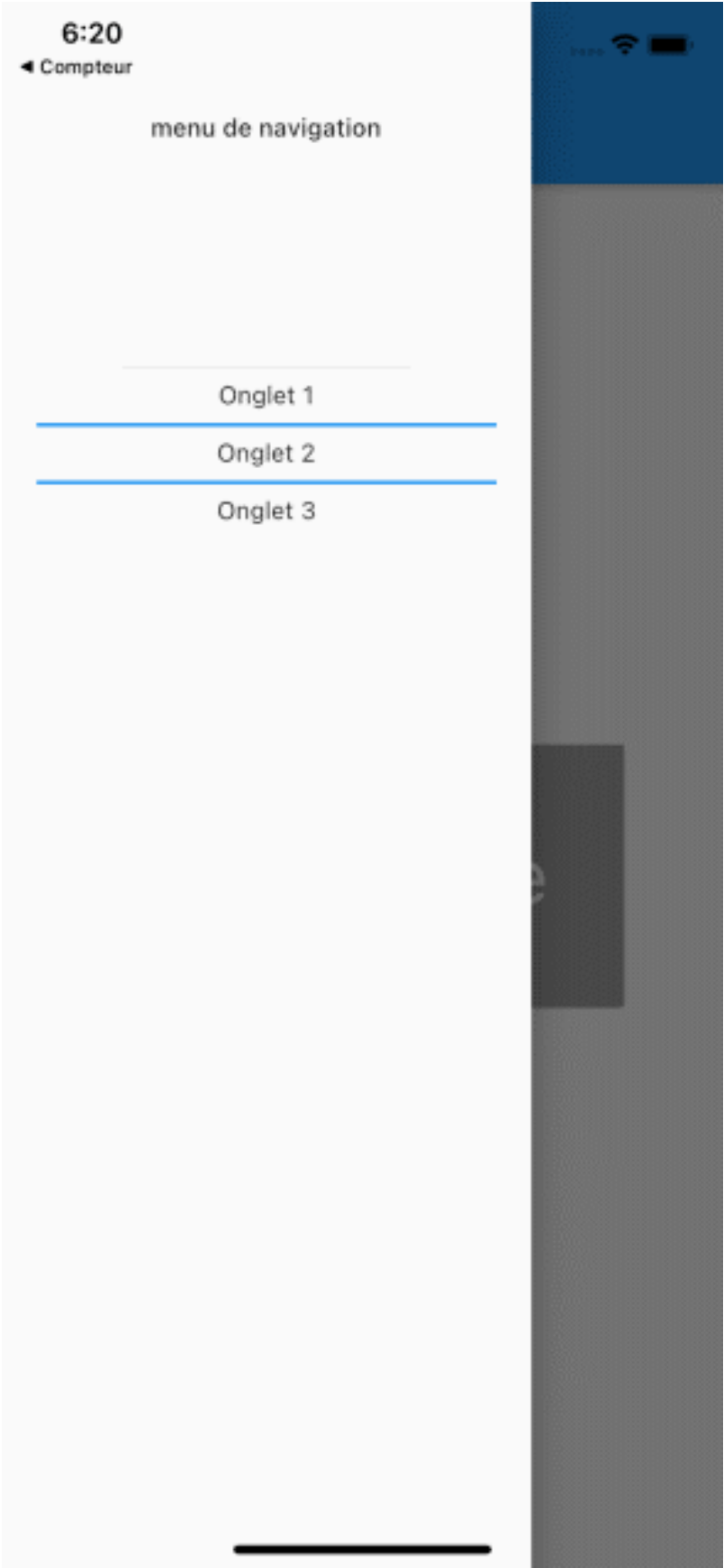
- `color` : la couleur du trait
- `indent` : son indentation par rapport à la marge de gauche
- `endIndent` : son indentation par rapport à la marge de droite
- `thickness` : l'épaisseur du trait

On définit un widget « *Drawer* » dont les onglets sont séparés par des traits en utilisant le widget « *Divider* ».

```

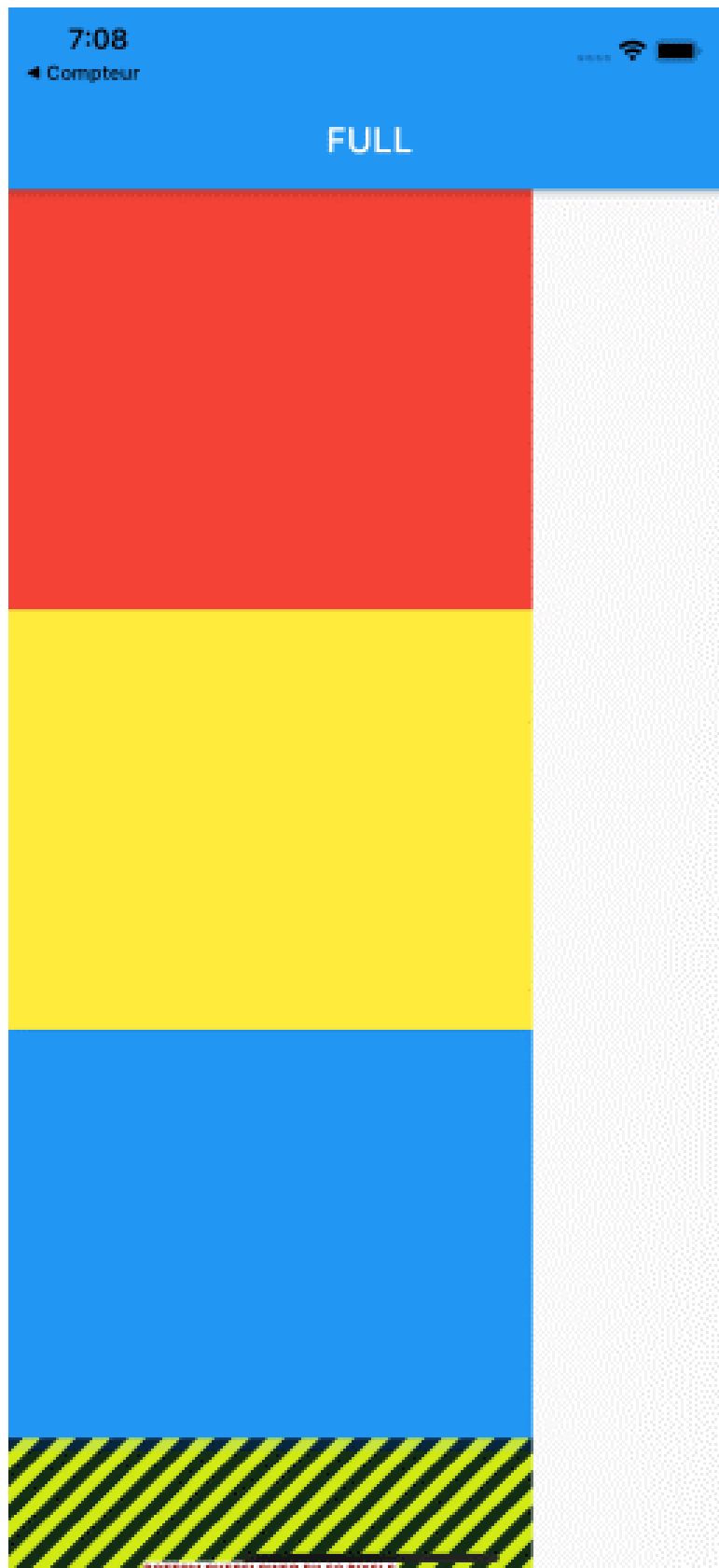
1 import 'package:flutter/material.dart';
2 class DrawerDivider extends StatelessWidget {
3   @override
4   Widget build(BuildContext context) {
5     return Drawer(
6       child: Column(
7         children: [
8           DrawerHeader(child: Text('menu de navigation'))
9         ],
10        Text("Onglet 1"),
11        Divider(
12          color: Colors.blue,
13          indent: 20,
14          endIndent: 20,
15          thickness: 2,
16        ),
17        Text("Onglet 2"),
18        Divider(
19          color: Colors.blue,
20          indent: 20,
21          endIndent: 20,
22          thickness: 2,
23        ),
24        Text("Onglet 3"),],
25      )
26    );
27  }
28 }
29 .

```



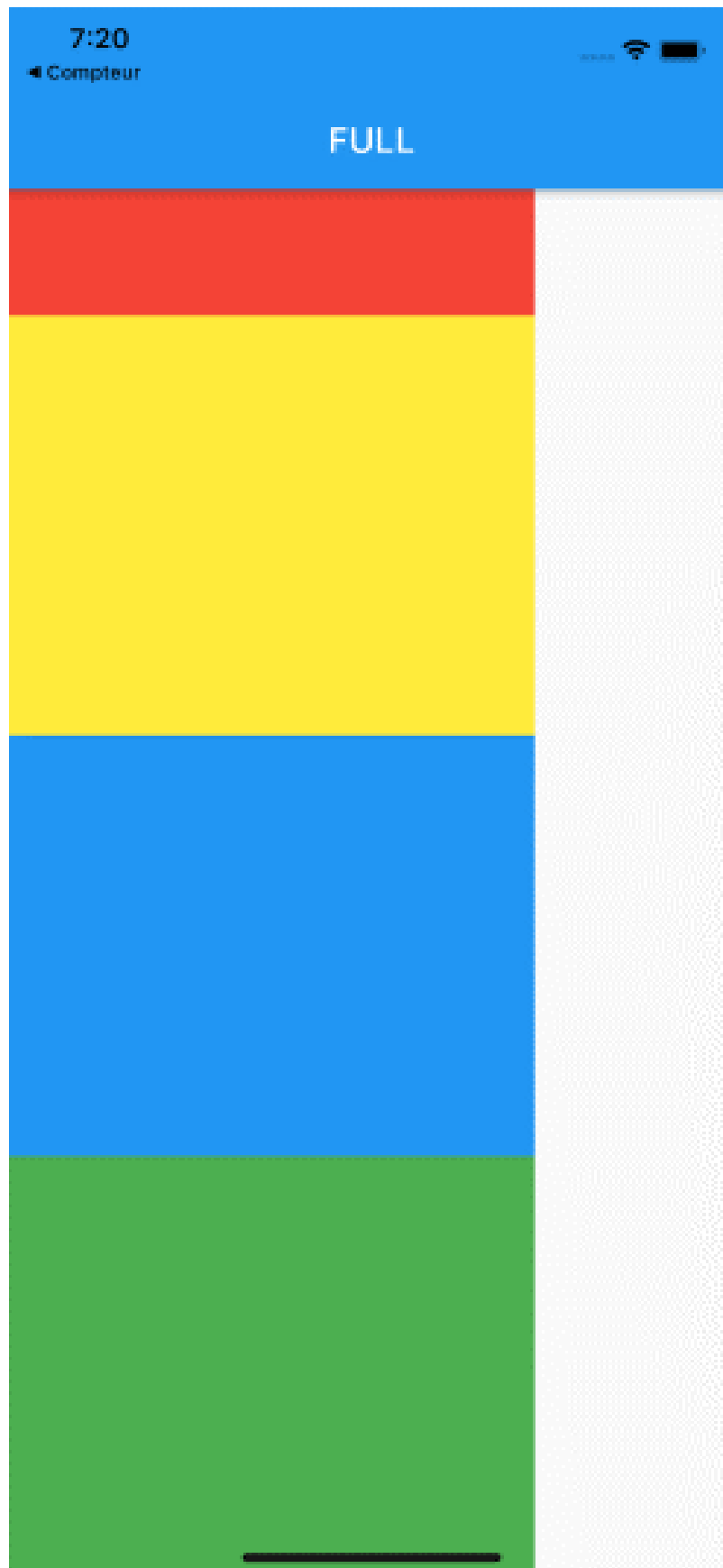
SingleChildScrollView

Lorsque l'on empile des éléments dans un widget « *Column* », il arrive que nos éléments dépassent de l'écran, car celui-ci n'est pas assez grand pour tout afficher. On est alors confronté à un débordement (overflow) d'un certain nombre de pixels. L'empilement d'éléments est trop grand par rapport à son contenant.



Le widget « *SingleChildScrollView* » va permettre de résoudre ce problème. Ce widget va être scrollable et va dérouler la colonne. La « *singleChildScrollView* » va englober le widget « *Column* ».

```
1 SingleChildScrollView(  
2   child: Column(  
3     children: [  
4       Container(  
5         color: Colors.red,  
6         height: 240,  
7         width: 300,  
8       ),  
9       Container(  
10        color: Colors.yellow,  
11        height: 240,  
12        width: 300,  
13      ),  
14      Container(  
15        color: Colors.blue,  
16        height: 240,  
17        width: 300,  
18      ),  
19      Container(  
20        color: Colors.green,  
21        height: 240,  
22        width: 300,  
23      ),  
24    ],  
25  ),  
26 ),  
27 )
```

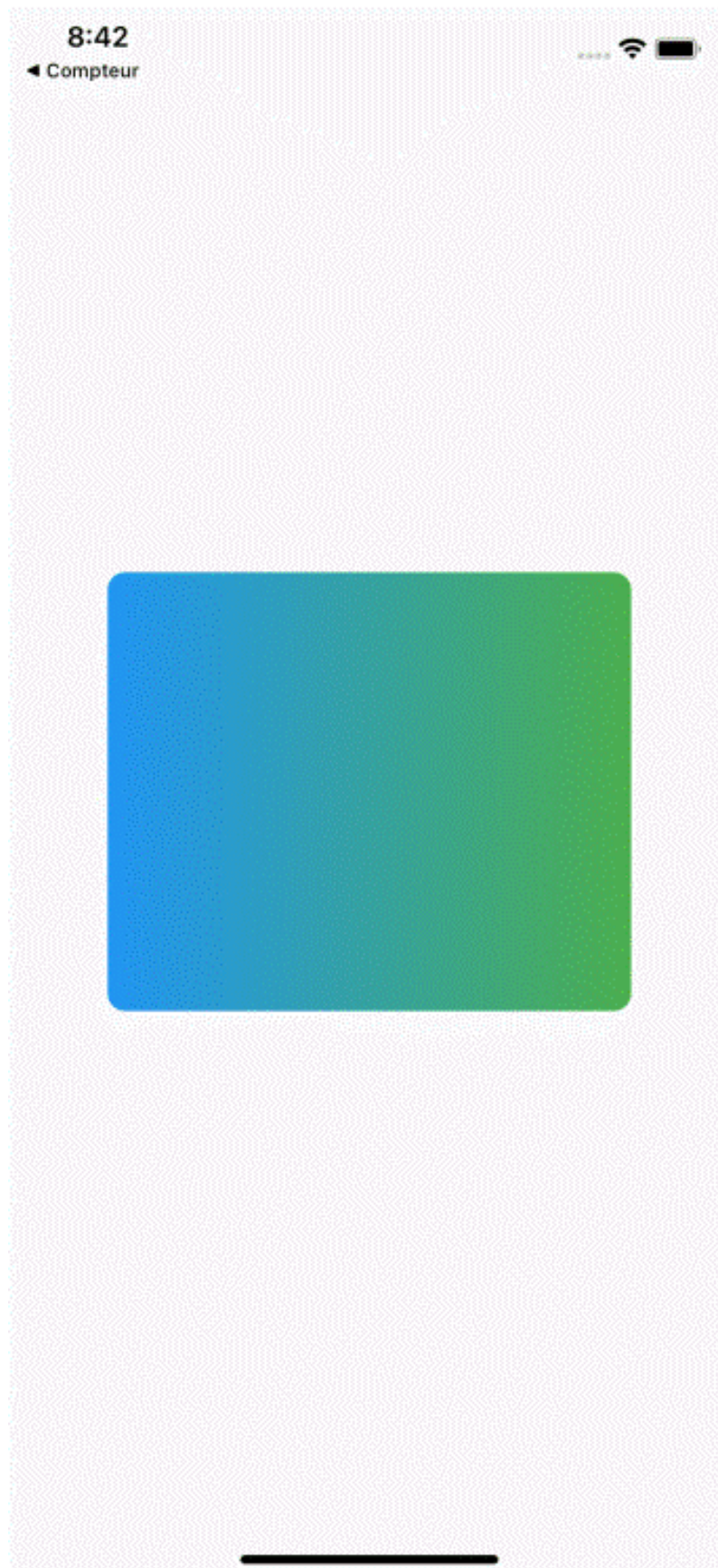


BoxDecoration

Le widget « *BoxDecoration* » permet de styliser une boîte, un container, par exemple à travers sa propriété *decoration*. Celle-ci va prendre un widget « *BoxDecoration* ». Le widget a les propriétés suivantes :

- `borderRadius` : spécifier les arrondis au container
- `gradient` : spécifier un dégradé en arrière-plan de notre container. Cette propriété « *gradient* » pourra prendre le widget « *LinearGradient* »

```
1 child: Container(  
2   height: 250,  
3   width: 300,  
4   decoration: BoxDecoration(  
5     borderRadius: BorderRadius.circular(10),  
6     gradient: LinearGradient(colors:[  
7       Colors.blue,  
8       Colors.green]  
9   )  
10 ),  
11 ),
```



Exercice : Quiz

Question 1

Quel mot clé permet d'intégrer un widget ?

- ☐ include
- ☐ import

Question 2

Quel widget permet de définir un cercle rempli ?

- ☐ CircularAvatar
- ☐ RoundedCircle
- ☐ CircleBox

Question 3

À quoi sert la propriété `borderRadius` dans le widget « *BoxDecoration* » à l'intérieur d'un container ?

- ☐ À définir l'épaisseur à la bordure
- ☐ À définir la longueur de la bordure
- ☐ À définir l'angle des arrondis du container

Question 4

Quel widget permet de remédier au dépassement de contenu à l'intérieur d'un écran ?

- ☐ scrollView
- ☐ listView
- ☐ singleChildScrollView

Question 5

Quel widget permet de faire une ligne de séparation ?

- ☐ Divider
- ☐ Spacer
- ☐ Separator

II. États

StatefulWidget

La gestion de l'état est un point crucial dans le développement d'une application. Jusqu'à présent, nos widgets héritent de « *StatelessWidget* ». Les widgets peuvent hériter de « *StatefulWidget* ». Ces deux héritages permettent de définir un comportement au « *class* » ou « *widget* ».

Un widget qui hérite de « *StatefulWidget* » va écouter l'état de ses variables. Le widget va se mettre à jour lorsque l'état des variables va changer.

On crée un widget « *MonInteractifPage* » qui sera un « *StatefulWidget* ». On le déclare dans un fichier séparé « *MonInteractif_page.dart* ».

```
1 class MonInteractifPage extends StatefulWidget {
```

```
2 }
```

Le widget se déclare comme un « *StatelessWidget* », sauf qu'il va hériter de « *StatefulWidget* ». Pour fonctionner, il va avoir besoin d'une autre classe qui va gérer l'état. Cette classe a le nom du widget avec le suffixe « *State* ».

```
1 class MonInteractifPageState extends State<InteractifPage> {
2   @override
3   Widget build(BuildContext context) {
4     return Container();
5   }
6 }
```

Pour que le widget `MonInteractifPage` écoute l'état, il faut surcharger la fonction `createState` à l'intérieur de la class « *MonInteractifPage* » qui va retourner « *MonInteractifState* ».

```
1 import 'package:flutter/material.dart';
2 class MonInteractifPage extends StatefulWidget {
3   @override
4   State<StatefulWidget> createState() {
5     // TODO: implement createState
6     return MonInteractifState();
7   }
8 }
```

On peut écrire la fonction dans une syntaxe plus concise :

```
1 @override
2 MonInteractifPageState createState() => MonInteractifPageState();
```

On écrit le constructeur du widget dans la class « *MonInteractifPageState* » :

```
1 class MonInteractifPageState extends State<InteractifPage> {
2   @override
3   Widget build(BuildContext context) {
4     return Scaffold(
5
6     appBar: AppBar(title: Text("StateFull"),),
7     body: Center(
8       child: Text("Statefull widget"),
9     ),
10  );
11 }
```

On retourne dans le fichier `main.dart`. Dans le widget « *materialApp* », au niveau de la propriété `home`, on va passer le widget créé en argument `MonInteractifPage`.

```
1 class MyApp extends StatelessWidget {
2   // This widget is the root of your application.
3   @override
4   Widget build(BuildContext context) {
5     return MaterialApp(
6       title: 'Flutter Demo',
7       theme: ThemeData(
8         primarySwatch: Colors.blue,
9       ),
10    home: MonInteractifPage()
11  );
12 }
13 }
```



Pour gérer le « *state* », deux fonctions sont nécessaires. Il faut les ajouter dans la class `MonInteractifPageState`:

- `initState` : tout ce qui va se faire pendant l'initialisation du widget.
- `dispose` : tout ce qui va se faire quand le widget sera supprimé.

```
1 @override
2 void initState() {
3   // TODO: implement initState
4   super.initState();
5 }
6
7 @override
8 void dispose() {
9   // TODO: implement dispose
10  super.dispose();
11 }
```

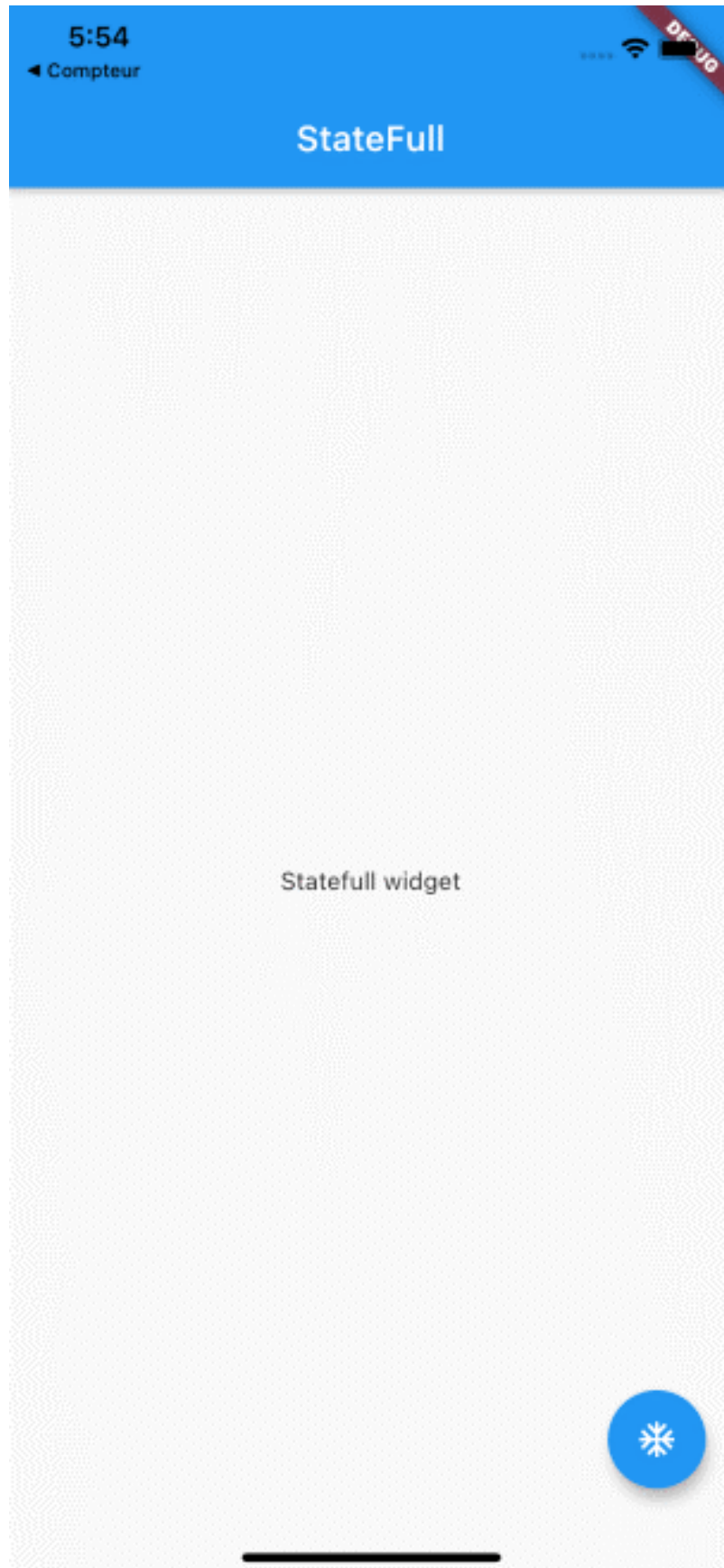
Le widget « `FloatingActionButton` »

Le « *FloatingActionButton* » est le premier widget interactif. Il est présent dans de nombreuses applications, comme les applications de gestion de boîtes mails.

On va le mettre dans le « *scaffold* » du widget « *MonInteractifPage* ». Il va avoir :

- Une propriété `onPressed`, qui va exécuter une fonction lors d'un appui sur le bouton. Dans la documentation, cette fonction est appelée « *Void Callback* ».
- Une propriété `child`, qui permet d'attacher une icône à l'intérieur du bouton.

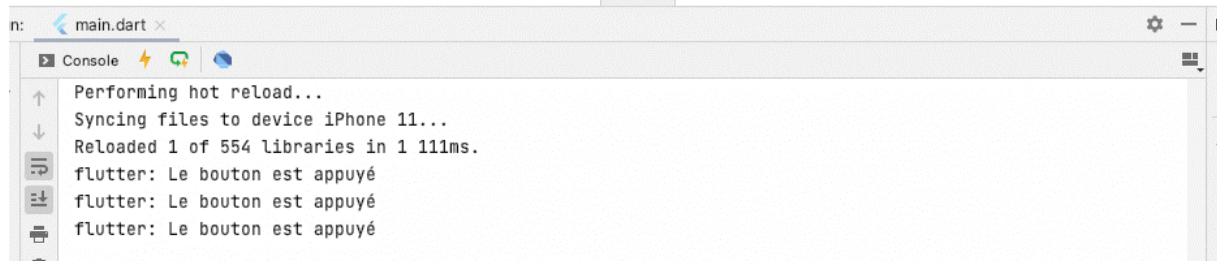
```
1 return Scaffold(
2   appBar: AppBar(title: Text("StateFull"),),
3   floatingActionButton: FloatingActionButton(
4     onPressed: (){
5
6     },
7     child: Icon(Icons.build),
8   ),
9   body: Center(
10    child: Text("Statefull widget"),
11  ),
12 );
13 }
```

Dans un premier temps, on va générer un affichage pour voir que l'interaction est bien prise en compte quand on appuie sur le bouton. On utilise la fonction `print` pour afficher un message.

```
1 onPressed: () {
2   print("Le bouton est appuyé");
3 },
```

On visualise l'appui du bouton grâce à l'affichage dans la console :



On va vouloir faire une opération plus intéressante qu'un simple « *affichage* ».

On va définir une variable `appBarColor` de type `Color` pour la barre et `textColor` de type `Color` pour la couleur du texte.

```
1 Color appBarColor = Colors.green;
2 Color textColor = Colors.red;
```

Le widget « *appBar* » dans le « *scaffold* » permet de configurer la couleur de la barre et du texte. Dans la fonction `onPressed()`, on va modifier la couleur de la barre du haut avec une condition ternaire. Lorsqu'on change une variable, il faut notifier le state avec la fonction `setState()`. La fonction `setState` va vérifier l'état et modifier celui-ci par rapport au comportement de l'utilisateur.

```
1 @override
2 Widget build(BuildContext context) {
3   return Scaffold(
4     appBar: AppBar(title: Text("StateFull"),
5       backgroundColor: appBarColor,),
6     floatingActionButton: FloatingActionButton(
7       onPressed: () {
8         print("Le bouton est appuyé");
9         setState(() {
10           appBarColor = (appBarColor == Colors.green) ? Colors.red : Colors.green;
11         });
12       },
13       child: Icon(Icons.ac_unit),
14     ),
15     body: Center(
16       child: Text("Statefull widget",
17         style: TextStyle(
18           color: textColor
19         ),),
20     ),
21   );
22 }
```

On peut faire une fonction pour cette mise à jour du state dans la propriété `onPressed`. Cette fonction s'appelle `changerCouleur()`.

```
1 changerCouleur() {
2   setState(() {
3     appBarColor = (appBarColor == Colors.green) ? Colors.red : Colors.green;
4   });
5 }
```

Dans le scaffold, la propriété `FloatingActionButtonLocation` va permettre de positionner le bouton à droite, à gauche ou au centre.

```
floatingActionButtonLocation:FloatingActionButtonLocation.centerDocked
```

Le widget « `TextButton` »

Dans le `body` de notre widget, on va ajouter un widget `Column`. Ce dernier aura des enfants (*children*), dont un widget `TextButton`.

```
1 body: Center(
2   child: Column(
3     children: [
4       TextButton(onPressed: null, child: Text("je suis un text button"))
5     ],
6   ),
7 ),
```

Dans la class « `MonInteractifPageState` », on ajoute une variable de type booléen :

```
bool textButtonPressed = false;
```

On définit une fonction pour changer le texte de la barre du haut. Si le bouton n'est pas pressé, le texte est : « *Je suis inactif* ». Sinon, le texte est : « *Mon Interactif* ». Cette fonction va renvoyer un `string`.

```
1 String changerAppText(){
2   return (textButtonPressed)? "je suis inactif" : "Mon Interactif";
3 }
```

On définit une fonction pour mettre à jour le `state`. Elle va appeler la fonction `setState()`. Cette fonction va faire passer la variable « `textButtonPressed` » à `true` si elle est à `false`. Si elle est à `true`, elle passe à `false`.

```
1 MettreAJourAppBar(){
2   setState(() {
3     textButtonPressed = !textButtonPressed;
4   });
5 }
```

On passe la fonction `MettreAJourAppBar` à la propriété `onPressed` de `textButton`.

```
1 TextButton(onPressed: MettreAJourAppBar, child:
2   Text(changerAppText()))
```

On peut changer la couleur de `background` de notre bouton en modifiant son `style`.

```
1 TextButton(onPressed: MettreAJourAppBar, child:
2   Text(changerAppText(),
3   style: TextStyle(
4     color: Colors.white,
5     backgroundColor: Colors.blue
6   ),
```

Le widget « `ElevationButton` »

Il ressemble au widget « `TextButton` » avec un effet visuel « *d'élévation* ». Il a une propriété `onLongPressed` pour gérer un appui long sur le bouton. On peut aussi lui changer son `style` : sa couleur, son élévation, et sa couleur de fond.

```
1 ElevatedButton(onPressed: (){
2   print("Salut");
3 }, child:const Text("Elevated"),
4 onLongPress: (){
5   print("appui loooooong");
6 },
```

```

7 style: ElevatedButton.styleFrom(
8
9   primary: Colors.green,
10  elevation: 20,
11  shadowColor: Colors.red
12 ),
13 ),

```

Le widget « TextField »

Le widget `TextField` est un champ de saisie d'un texte. Comme tous les widgets, il a de nombreuses propriétés permettant de modifier son aspect.

Il a les propriétés suivantes :

- `obscureText` : de type booléen pour cacher le texte.
- `decoration` : pour configurer l'aspect du `textField`.

La propriété « *decoration* » est associée à une instance de type `InputDecoration` qui prend en paramètres dans son constructeur :

- `hintText` : le texte par défaut,
- `border` : pour spécifier la forme de l'entourage du `textField`.
- `keyboardType` : pour spécifier le type de clavier (*phone, number, email adress*).
- `onChanged` : quand le texte va changer. On va récupérer la valeur du champ pour modifier le « *state* ».
- `onSubmit` : quand on appuie sur le bouton « *valider* » du clavier.

Avant la définition d'un `textField`, il faut déclarer une variable pour stocker la saisie de l'utilisateur.

```

1 String prenom = "";
2
3 TextField(
4   obscureText: false,
5   decoration: InputDecoration(
6     hintText: "Entrez votre prénom",
7     border: OutlineInputBorder(
8       borderRadius: BorderRadius.circular(20)
9     )
10  ),
11  keyboardType: TextInputType.emailAddress,
12  onChanged: (newString){
13    setState(() {
14      prenom = newString;
15    });
16  },
17 ),

```

On a vu que la saisie en temps réel au clavier se fait avec la méthode `onChanged`.

Il existe une deuxième façon de procéder. Ici, on va mettre en place un contrôleur qui fonctionne avec un « *pattern d'observation* ». On ajoute un `textEditingController` au widget.

```

1 late TextEditingController controller;

```

Le mot clé `late` indique que le contrôleur ne sera pas `null` plus tard. Il faut l'initialiser et réaliser les abonnements au contrôleur dans `initState` avec la fonction `addListener`.

```

1 @override
2 void initState() {
3   // TODO: implement initState
4   super.initState();

```

```
5 controller = TextEditingController();
6 controller.addListener(_montrerLeTexteSaisie);
7 }
```

Dans le `dispose`, on arrête le `controller`. On retire le widget. Cela libère de la mémoire.

```
1 @override
2 void dispose() {
3   controller.dispose();
4   super.dispose();
5 }
```

Après les étapes de configuration passées, il faut écrire la méthode privée nommée `_montrerLeTexteSaisie`. Elle va mettre à jour la variable `prenom` avec `setState`.

```
1 void _montrerLeTexteSaisie() {
2   setState(() {
3     prenom = controller.text;
4   });
5 }
```

Ensuite, on déclare le `textField` de la façon suivante :

```
1 TextField(
2   controller: controller,
3   decoration: InputDecoration(
4     hintText: "Entrez votre prenom"
5   ),
6 ),
7 Text(prenom)
```

Les deux façons donnent le même résultat dans la gestion du `TextField`. Avec la deuxième façon, il n'est plus nécessaire d'écrire la fonction pour la propriété `onChanged`. Elle permet un meilleur contrôle sur l'évolution du champ.

Le widget « switch »

Le widget `switch` est un interrupteur. Son constructeur prend deux paramètres obligatoires :

- Une valeur booléenne.
- La propriété `onChanged` qui va contenir la méthode à appeler lorsqu'on appuie sur le `switch`.

On définit alors ce booléen :

```
1 bool switchValue = true;
```

Et on va construire notre `switch` de cette façon :

```
1 Switch(value: switchValue, onChanged: (bool) {
2   setState(() {
3     switchValue = bool;
4   });
5 })
```

En actionnant l'interrupteur, on va mettre à jour `switchValue`.

On va aligner l'interrupteur avec un texte. En fonction de l'état de l'interruption, un widget `Text` affichera « *iOS* » ou « *Android* ».

```
1 Padding(
2   padding: const EdgeInsets.all(10.0),
3   child: Row(
4     children: [
5       Text((switchValue) ? "iOS" : "Android"),
6       Switch(value: switchValue, onChanged: (bool) {
```

```

7      setState(() {
8          switchValue = bool;
9      });
10     })
11 ],
12 ),
13 )

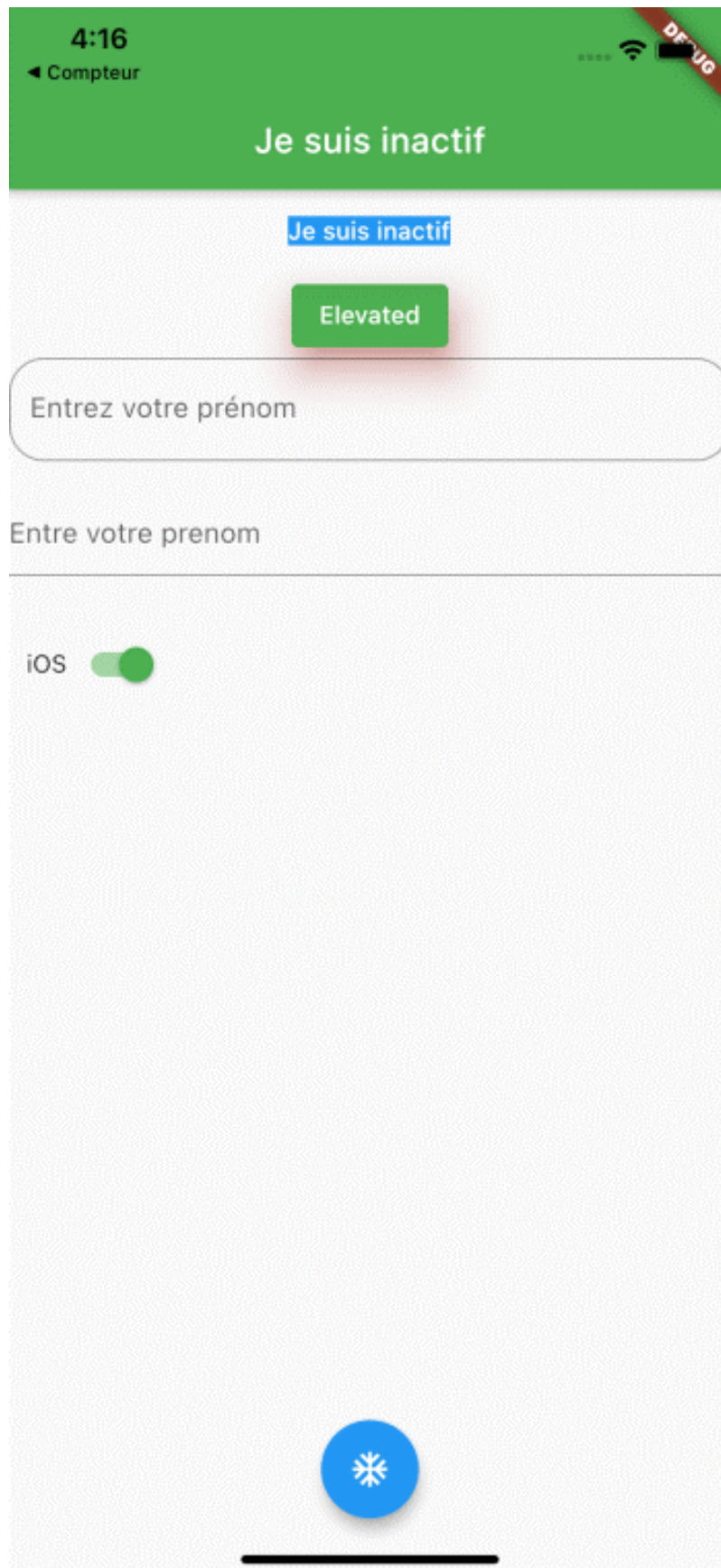
```

On peut modifier la propriété `activeColor` pour modifier l'aspect de notre switch.

```

1 Switch(value: switchValue,
2   activeColor: Colors.green,
3   onChanged: (bool) {
4     setState(() {
5       switchValue = bool;
6     });
7 })

```



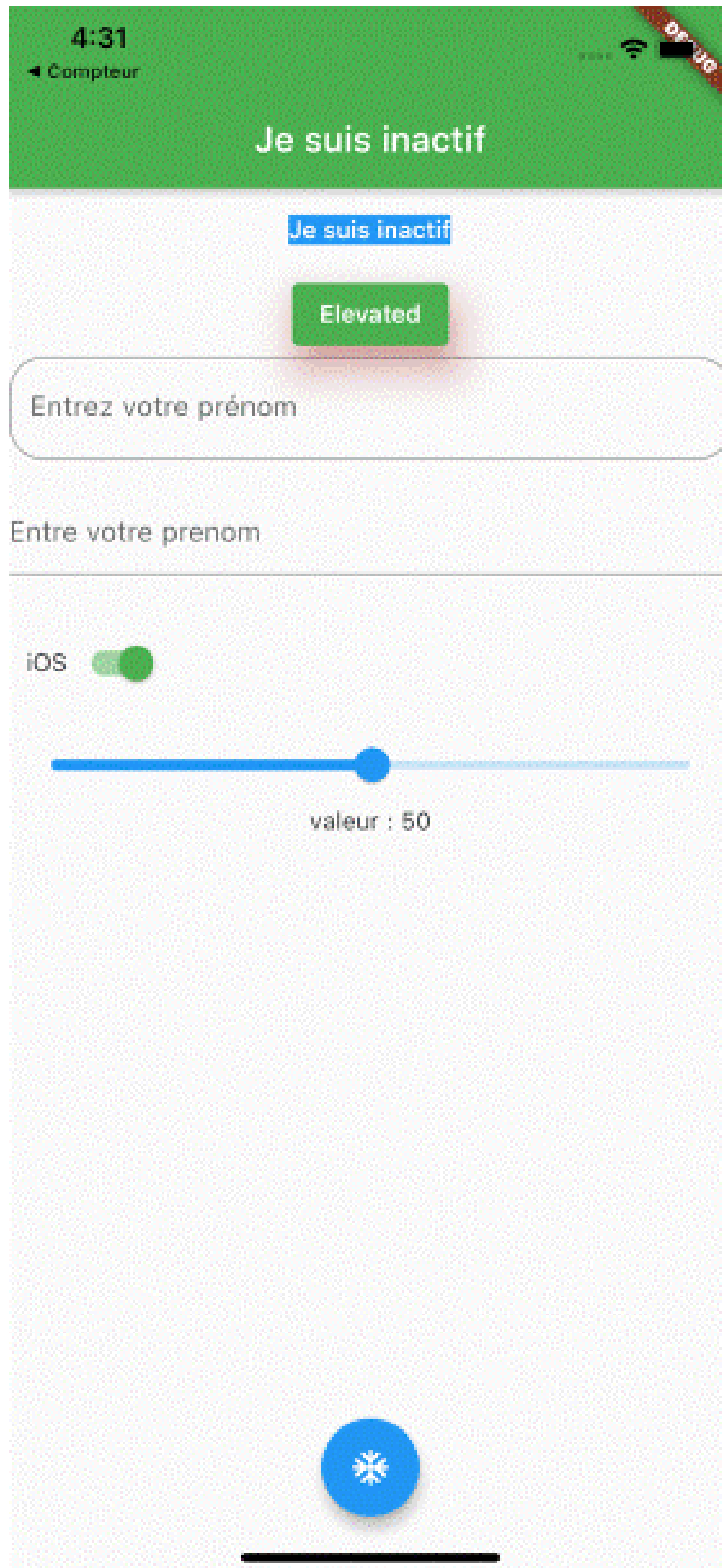
Le widget « Slider »

Le widget « *Slider* » est un autre composant que l'on trouve dans des formulaires. Il permet à l'utilisateur de choisir une valeur sur une échelle. Son constructeur prend une valeur de type double et une propriété `onChanged`. Il faut aussi définir une propriété `min` et `max`.

```

1 Slider(value: sliderValue,
2     min: 0,
3     max: 100,
4     onChanged: ((newValue){
5   setState(() {
6     sliderValue = newValue;
7     print(sliderValue);
8   }));
9 })),
10 Text("valeur : ${sliderValue.toInt()}")

```

Le widget « checkBox »

Le widget « *checkBox* » se présente comme une case à cocher. Il pourra être utilisé pour une liste de courses. Chaque élément de la liste est associé à un booléen.

On définit une liste de courses de la façon suivante :

```
1 Map<String, bool> listeDeCourses = {
2   "Lait":false,
3   "Salade":true,
4   "Café":false};
```

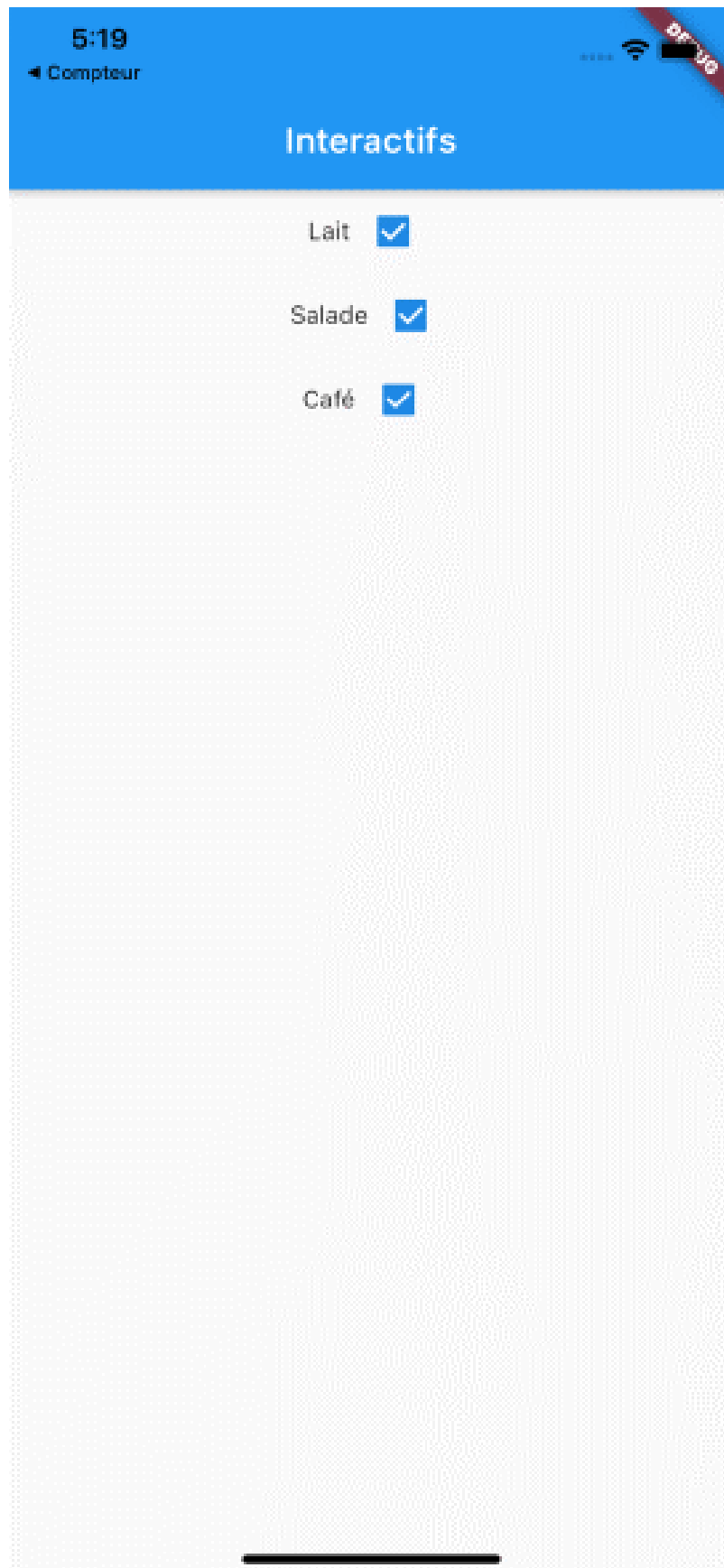
On définit une fonction `checks` qui retourne une « *Column* » d'éléments à cocher.

```
1 Column checks() {
2   List<Widget> items = [];
3   listeDeCourses.forEach((course, acheter) {
4     Widget row = Row(
5       mainAxisAlignment: MainAxisAlignment.center,
6       mainAxisAlignment:MainAxisSize.max ,
7       children: [
8         Text(course),
9         Checkbox(value: acheter, onChanged: ((newValue){
10           setState(() {
11             listeDeCourses[course] = newValue ?? false;
12           });
13         })))
14     ],
15   );
16   items.add(row);
17 }
18 }
19 }
20 return Column(children: items);
21 }
```

La fonction construit une liste de widgets, items de type « *Row* » contenant l'intitulé de l'objet à acheter et d'une « *checkBox* » (case à cocher). On itère sur chaque élément de la liste. À chaque fois, un widget « *Row* » est ajouté à la liste. Enfin, la fonction renvoie un Widget « *column* » contenant les enfants de la liste d'items.

Au niveau de body du scaffold, on met la fonction `checks ()` .

```
1 @override
2 Widget build(BuildContext context) {
3   return Scaffold(
4     appBar: AppBar(title: Text("Interactifs"),
5   ),
6     body: Center(
7       child: checks(),
8     ),
9   ),
10 );
11 );
12 }
```



Exercice : Quiz

Question 1

Quelle est la spécificité d'un « *StatefulWidget* » ?

- ☐ Il possède toutes les fonctions d'un widget
- ☐ Il ne sera jamais reconstruit
- ☐ Il possède un état

Question 2

Quelle fonction va créer le state d'un « *StatefulWidget* » ?

- ☐ generateState()
- ☐ initState()
- ☐ createState()

Question 3

Quelle fonction permet de mettre à jour le state d'un « *StatefulWidget* » ?

- ☐ updateState
- ☐ setState
- ☐ putState

Question 4

Quel widget permet à l'utilisateur de saisir un texte ?

- ☐ textView
- ☐ TextField
- ☐ textButton

Question 5

Quel widget a le comportement d'un interrupteur ?

- ☐ Interrupt
- ☐ Switch

III. Essentiel

Dans ce cours, nous avons vu que l'on pouvait découper notre projet en plusieurs fichiers. Et nous avons exploré les widgets suivants :

- **CircularAvatar** : ce widget est un cercle à l'intérieur duquel on peut mettre une couleur en arrière-plan, une image d'arrière-plan, un texte d'arrière-plan.
- **Divider** : ce widget permet de créer une ligne séparatrice entre des widgets. Son utilisation est courante pour séparer les éléments dans une liste.
- **SingleChildScrollView** : ce widget permet de résoudre le problème quand l'écran est trop petit pour tout le contenu.
- **BoxDecoration** : ce widget permet de styliser un Container (coins arrondis - dégradé).

Ensuite, nous avons vu les deuxièmes sortes de widgets qui sont les « *StatefulWidget* ». Ces widgets vont écouter l'état des variables. Les composants interactifs, tels des boutons, vont faire appel à une fonction « *setState* » qui va mettre à jour l'état.

On a vu les widgets interactifs suivants :

- **FloatingActionButton** : le bouton qui se trouve en bas de l'écran.
- **TextButton** : un bouton qui est un simple texte.
- **ElevationButton** : identique au TextButton, mais possède une élévation.
- **Switch** : un interrupteur qui va prendre une valeur vraie ou fausse.
- **Slider** : permet de choisir une valeur dans un intervalle.
- **CheckBox** : permet de cocher des items dans une liste.

IV. Auto-évaluation

A. Exercice :

Dans cet exercice, nous allons réaliser un petit formulaire à l'aide des différents widgets découverts dans ce cours. Ce formulaire est un « *StatefulWidget* ». On le nommera « *MonFormulairePage* ». Il dispose d'un état qui va évoluer avec les interactions de l'utilisateur. Pour l'essentiel, ces widgets vont avoir comme propriété une fonction qui mettra à jour l'état de « *Mon Formulaire* ». Les widgets à intégrer dans notre formulaire sont des « *textField* », un « *switch* », des « *checkBox* », des boutons. Pour le *textField*, il faudra implémenter son fonctionnement de deux manières. En bas de l'écran, une zone va afficher la synthèse des informations saisies. La zone d'affichage utilisera des widgets de base que nous avons vus dans la partie 1. Il faudra combiner des widgets « *Row* » et « *Column* » pour disposer correctement les éléments.

3:06

Mon Formulaire

Martin

Loic

Homme

Cochez les Langages validés

Php

Flutter

Javascript

Prévisualisation

Nom : Martin

Prenom : Loic

Homme

Langages validés

Php

Flutter

Question 1

Créer un nouveau projet « *mondefi* ». Construire le widget « *StatefulWidget* » ayant pour nom « *Formulaire* » avec les classes de base et fonctions. Il sera dans un fichier séparé. L'appeler dans le fichier main.

Question 2

Créez un « *scaffold* » pour le squelette de votre application. À l'intérieur du body, créez une « *Column* ». Cette « *Column* » est à l'intérieur d'un widget « *singleChildScrollView* » pour ne pas avoir de problème d'affichage. La « *Column* » contient deux blocs (containers) :

- Un bloc pour les champs de saisie avec une couleur de fond grise. Le bloc occupe $\frac{2}{3}$ de l'écran.
- Un bloc pour la prévisualisation. Il occupe $\frac{1}{3}$ de l'écran. Sa couleur de fond est verte.

Question 3

Ajoutez deux zones de texte dans le premier bloc. Ajoutez un style arrondi aux zones de texte. Mettez un texte.

Question 4

Ajoutez à la première zone de texte une méthode « *onChanged* » pour récupérer la chaîne de caractères saisie. Elle est ensuite affichée dans la zone du bas.

Question 5

Ajoutez un « *controller* » pour gérer le second textField. Affichez le texte saisi dans la zone de prévisualisation.

Question 6

Ajoutez une « *checkbox* » pour les cours validés. Faire une fonction pour afficher les résultats cochés dans le cadre de prévisualisation.

Question 7

Vérifiez votre code.

Question 8

Quel est le widget complet ?

B. Test**Exercice : Quiz**

Question 1

Quel widget peut-on utiliser quand le contenu dépasse de l'écran ?

- ☐ SingleChildScrollView
- ☐ listView
- ☐ Scrollview

Question 2

Quel widget permet de faire des séparations entre les widgets ?

- ☐ Liner
- ☐ Spacer
- ☐ Divider

Question 3

Quelle propriété permet de styliser un textField ?

- ☐ textFieldProperty
- ☐ textFieldDecoration
- ☐ decoration

Question 4

Quelle propriété permet de définir la fonction lorsqu'on change l'état du widget « *Switch* » ?

- ☐ onSubmit
- ☐ onChanged

Question 5

Quel contrôleur peut gérer un textField ?

- ☐ textController
- ☐ TextEditingController
- ☐ ControllerOfTextField

Solutions des exercices

Exercice p. 13 Solution n°1

Question 1

Quel mot clé permet d'intégrer un widget ?

- ☐ include
- ☒ import



Le mot clé « *import* » permet d'intégrer un widget qui est dans un fichier séparé. En répartissant son code en plusieurs fichiers, le projet est plus facile à comprendre.

Question 2

Quel widget permet de définir un cercle rempli ?

- ☒ CircularAvatar
- ☐ RoundedCircle
- ☐ CircleBox



« *CircularAvatar* » est un widget qui permet de définir un cercle. Celui-ci peut prendre en arrière-plan : un fond uni, une image, un texte.

Question 3

À quoi sert la propriété `borderRadius` dans le widget « *BoxDecoration* » à l'intérieur d'un container ?

- ☐ À définir l'épaisseur à la bordure
- ☐ À définir la longueur de la bordure
- ☒ À définir l'angle des arrondis du container



Elle sert à définir l'angle des arrondis du container. `BorderRadius` est une propriété de « *BoxDecoration* ». Ainsi, on donne un rayon de bordure au container.

Question 4

Quel widget permet de remédier au dépassement de contenu à l'intérieur d'un écran ?

- ☐ `scrollView`
- ☐ `listView`
- ☒ `singleChildScrollView`



`SingleChildScrollView` permet d'y remédier quand le contenant ne peut pas s'afficher en intégralité. Ce widget crée un « *scroll* » à la manière d'une liste. Le « *scroll* » peut être horizontal ou vertical.

Question 5

Quel widget permet de faire une ligne de séparation ?

- ☒ `Divider`
- ☐ `Spacer`
- ☐ `Separator`




« *Divider* » permet de faire une ligne de séparation. On l'utilise dans le drawer (menu) pour séparer les onglets.

Exercice p. 28 Solution n°2


Question 1

Quelle est la spécificité d'un « *StatefulWidget* » ?

- ☐ Il possède toutes les fonctions d'un widget
- ☐ Il ne sera jamais reconstruit
- ☒ Il possède un état
-  La spécificité d'un « *StatefulWidget* » est de disposer d'un état. Le widget sera reconstruit quand son état va changer.


Question 2

Quelle fonction va créer le state d'un « *StatefulWidget* » ?

- ☐ `generateState()`
- ☐ `InitState()`
- ☒ `CreateState()`
-  La fonction « *createState* » va générer l'état du widget. Un exemple de déclaration est « *MonWidgetState createState() → MonWidgetState()* ».


Question 3

Quelle fonction permet de mettre à jour le state d'un « *StatefulWidget* » ?

- ☐ `updateState`
- ☒ `setState`
- ☐ `putState`
-  La fonction « *setState* » permet de mettre à jour l'état du widget. Si on n'utilise pas cette fonction, rien ne va se passer. Le widget ne changera pas.


Question 4

Quel widget permet à l'utilisateur de saisir un texte ?

- ☐ `textView`
- ☒ `TextField`
- ☐ `textButton`
-  Le widget « *TextField* » permet à l'utilisateur de saisir du texte. Ce widget dispose de la propriété « *decoration* » pour configurer son apparence.

Question 5

Quel widget a le comportement d'un interrupteur ?

- ☐ `Interrupt`
- ☒ `Switch`
-  Le widget « *Switch* » permet de proposer deux choix à l'utilisateur comme un interrupteur. Ce widget prend la valeur de « *vrai* » ou de « *faux* ». Cette valeur est de type booléenne.

Exercice p. Solution n°3

On crée un fichier « *monformulaire_page.dart* » pour le widget « *MonFormulairePage* ». On crée le widget « *MonFormulairePage* » avec les classes de base pour implémenter un « *StatefulWidget* ». Le widget est appelé dans le fichier « *main.dart* ».

Le fichier « *monformulaire_page.dart* » :

```

1 import 'package:flutter/material.dart';
2
3 class MonFormulairePage extends StatefulWidget {
4   @override
5   MonFormulairePageState createState() => MonFormulairePageState();
6 }
7
8 class MonFormulairePageState extends State<MonFormulairePage> {
9
10  @override
11  void initState() {
12    // TODO: implement initState
13    super.initState();
14  }
15
16  @override
17  void dispose() {
18    // TODO: implement dispose
19    super.dispose();
20  }
21
22  @override
23  Widget build(BuildContext context) {
24    return Container();
25  }
26 }

```

Le fichier « *main.dart* » :

```

1 import 'package:flutter/material.dart';
2 import 'package:monformulaire/monformulaire_page.dart';
3
4 void main() {
5   runApp(MyApp());
6 }
7
8 class MyApp extends StatelessWidget {
9   // This widget is the root of your application.
10  @override
11  Widget build(BuildContext context) {
12    return MaterialApp(
13      title: 'Flutter Demo',
14      theme: ThemeData(
15        primarySwatch: Colors.blue,
16      ),
17      home: MonFormulairePage()
18    );
19  }
20 }

```

Exercice p. Solution n°4

On intègre le scaffold dans notre widget « *MonFormulairePage* ». On met une « *appBar* », body qui contient deux containers. On récupère la taille de l'écran à partir du « *context* ». Le widget « *Column* » est à l'intérieur d'une « *SingleChildScrollView* ».

Le build du widget « *MonFormulaire* » :

```

1 @override
2 Widget build(BuildContext context) {
3   var size = MediaQuery.of(context).size;
4   return Scaffold(
5     appBar: AppBar(
6       title: Text("Mon Formulaire")
7     ),
8     body: SingleChildScrollView(
9       child: Column(
10        children: [
11          Container(
12            height: 2*size.height/3,
13            width: size.width,
14            color: Colors.black12,
15          ),
16          Container(
17            height: 1*size.height/3,
18            width: size.width,
19            color: Colors.green,
20          )
21        ],
22      ),
23    ),
24  );
25 };
26 }
27 }

```

Exercice p. Solution n°5

On crée deux zones de saisie de texte dans le container du haut : un pour le nom et un pour le prénom. On utilise les propriétés du widget adéquat pour le style de la zone de texte.

```

1 Container(
2   height: 2*size.height/3 - 50,
3   width: size.width,
4   child: Column(
5     children: [
6       Padding(
7         padding: const EdgeInsets.all(8.0),
8         child: TextField(
9           decoration: InputDecoration(
10            hintText: "Entrez votre nom",
11            border: OutlineInputBorder(
12              borderRadius: BorderRadius.circular(20)
13            )
14          ),
15        ),
16      ),
17      Padding(
18        padding: const EdgeInsets.all(8.0),
19        child: TextField(

```

```

20         decoration: InputDecoration(
21             hintText: "Entrez votre Prenom",
22             border: OutlineInputBorder(
23                 borderRadius: BorderRadius.circular(20)
24             )
25         ),
26     ),
27 )
28 ],
29 ),
30 color: Colors.black12,
31 )

```

Exercice p. Solution n°6

Dans notre « *class* » qui gère l'état, on ajoute une variable pour récupérer les caractères saisis. Ensuite, on implémente la fonction au niveau de la propriété « *onChanged* ». Au niveau de la zone du bas, on affiche le nom.

Le premier textField :

```

1  Padding(
2    padding: const EdgeInsets.all(8.0),
3    child: TextField(
4      decoration: InputDecoration(
5        hintText: "Entrez votre nom",
6        border: OutlineInputBorder(
7          borderRadius: BorderRadius.circular(20)
8        )
9      ),
10     onChanged: (newValue) {
11       setState(() {
12         nom = newValue;
13       });
14     },
15   ),
16 ),

```

Exercice p. Solution n°7

On crée un controller. On initialise le controller dans l'initialisation de l'état (*initState*) et on lui ajoute une fonction « *observation* ». On crée la fonction « *observatrice* ». On ajoute une variable pour stocker le prénom.

```

1  class MonFormulairePageState extends State<MonFormulairePage> {
2
3    late TextEditingController controller;
4    String nom = "";
5    String prenom = "";
6
7    @override
8    void initState() {
9      // TODO: implement initState
10     super.initState();
11     controller = TextEditingController();
12     controller.addListener(_montrerLeTexteSaisie);
13   }
14
15   @override

```

```

16 void dispose() {
17     // TODO: implement dispose
18     super.dispose();
19     controller.dispose();
20 }
21
22
23 void _montrerLeTexteSaisie(){
24     print("monter texte saisie");
25     setState(() {
26         prenom = controller.text;
27     });
28 }

```

Au niveau du textField du prénom, on ajoute une propriété « *controller* ».

```

1 TextField(
2   controller: controller,
3   decoration: InputDecoration(
4     hintText: "Entrez votre Prenom",
5     border: OutlineInputBorder(
6       borderRadius: BorderRadius.circular(20)
7     )
8   ),
9 ),

```

Le container pour afficher le champ de texte saisi :

```

1 Container(
2   height: 1*size.height/3+50,
3   width: size.width,
4   color: Colors.green,
5   child: Column(
6     children: [
7       Text("Prévisualisation"),
8       Text("Nom : ${nom}"),
9       Text("Prenom : ${prenom}")
10    ],
11  ),
12 ),
13 )

```

Exercice p. Solution n°8

On rajoute une variable « *switchValue* ». On rajoute une cellule (Row) contenant le switch.

```

1 Padding(
2   padding: const EdgeInsets.all(8.0),
3   child: Row(
4     children: [
5       Text((switchValue) ? "Homme" : "Femme"),
6       Switch(value: switchValue,
7         activeColor: Colors.green,
8         onChanged: (bool) {
9           setState(() {
10             switchValue = bool;
11           });
12         }
13     ],

```

```
14 )
```

Au niveau du container de prévisualisation :

```
1 Container(
2   height: 1*size.height/3+50,
3   width: size.width,
4   color: Colors.green,
5   child: Column(
6     children: [
7       Text("Prévisualisation"),
8       Text("Nom : ${nom}"),
9       Text("Prenom : ${prenom}"),
10
11       Text((swichValue) ? "Homme" : "Femme"),
12
13     ],
14   ),
15 )
```

Exercice p. Solution n°9

On crée une map pour stocker une liste de courses.

```
1 Map<String, bool> listeDeCourses = {
2   "langage php":false,
3   "langage flutter":true,
4   "langage":false};
5 };
```

On crée une fonction qui fait une liste de checkBox.

```
1 Column checks() {
2   List<Widget> items = [];
3   listeDeCourses.forEach((course, acheter) {
4     Widget row = Row(
5       mainAxisAlignment: MainAxisAlignment.center,
6       mainAxisAlignment:MainAxisSize.max ,
7       children: [
8         Text(course),
9         Checkbox(value: acheter, onChanged: ((newValue){
10           setState(() {
11             listeDeCourses[course] = newValue ?? false;
12           });
13         })))
14     ],
15   );
16   items.add(row);
17
18
19 });
20 return Column(children: items);
21 }
```

On crée une fonction « *langageEtudie* » qui retourne les langages cochés dans la prévisualisation. Elle retourne un widget de type « *Column* ».

```
1 Column langageEtudie(){
2   List<Widget> items = [];
3   listeDeCourses.forEach((cours, validate) {
```

```

4   print(validate);
5   if (validate == true){
6     Widget row = Row(
7       mainAxisAlignment:MainAxisAlignment.center ,
8       children: [
9         Text(cours)
10      ],
11    );
12    items.add(row);
13  }
14  });
15  return Column(children:items);
16 }

```

Exercice p. Solution n°10

Le widget entier :

```

1
2 import 'package:flutter/material.dart';
3
4 class MonFormulairePage extends StatefulWidget {
5   @override
6   MonFormulairePageState createState() => MonFormulairePageState();
7 }
8
9 class MonFormulairePageState extends State<MonFormulairePage> {
10
11   Map<String, bool> listeDeCours = {
12     "Php":false,
13     "Flutter":true,
14     "Javascript":false};
15
16   bool swichValue = true;
17
18   late TextEditingController controller;
19   String nom = "";
20   String prenom = "";
21
22   @override
23   void initState() {
24     // TODO: implement initState
25     super.initState();
26     controller = TextEditingController();
27     controller.addListener(_montrerLeTexteSaisie);
28   }
29
30   @override
31   void dispose() {
32     // TODO: implement dispose
33     super.dispose();
34     controller.dispose();
35   }
36
37
38   void _montrerLeTexteSaisie(){
39     print("monter texte saisie");

```



```

40     setState(() {
41         prenom = controller.text;
42     });
43 }
44
45
46
47
48 @override
49 Widget build(BuildContext context) {
50     var size = MediaQuery.of(context).size;
51     return Scaffold(
52         appBar: AppBar(
53             title: Text("Mon Formulaire")
54         ),
55         body: SingleChildScrollView(
56             child: Column(
57                 children: [
58                     Container(
59                         height: 2*size.height/3 - 50,
60                         width: size.width,
61                         child: Column(
62                             children: [
63                                 Padding(
64                                     padding: const EdgeInsets.all(8.0),
65                                     child: TextField(
66                                         decoration: InputDecoration(
67                                             hintText: "Entrez votre nom",
68                                             border: OutlineInputBorder(
69                                                 borderRadius: BorderRadius.circular(20)
70                                             )
71                                         ),
72                                         onChanged: (newValue){
73                                             setState(() {
74                                                 nom = newValue;
75                                             });
76                                         },
77                                     ),
78                                 ),
79                                 Padding(
80                                     padding: const EdgeInsets.all(8.0),
81                                     child: TextField(
82                                         controller: controller,
83                                         decoration: InputDecoration(
84                                             hintText: "Entrez votre Prenom",
85                                             border: OutlineInputBorder(
86                                                 borderRadius: BorderRadius.circular(20)
87                                             )
88                                         ),
89                                     ),
90                                 ),
91                                 Padding(
92                                     padding: const EdgeInsets.all(8.0),
93                                     child: Row(
94                                         children: [
95                                             Text((switchValue) ? "Homme" : "Femme"),
96                                             Switch(value: switchValue,
97                                                 activeColor: Colors.green,

```

```

98         onChanged: (bool) {
99             setState(() {
100                 swichValue = bool;
101             });
102         })),
103     ],
104     ),
105     ),
106     Text("Cochez les Langues validés"),
107     Padding(
108         padding: const EdgeInsets.all(8.0),
109         child: checks(),
110     )
111 ],
112 ),
113 ),
114 color: Colors.black12,
115 ),
116 Container(
117     height: 1*size.height/3+50,
118     width: size.width,
119     color: Colors.green,
120     child: Column(
121         children: [
122             Text("Prévisualisation"),
123             Text("Nom : ${nom}"),
124             Text("Prenom : ${prenom}"),
125
126             Text((swichValue) ? "Homme" : "Femme"),
127             Text("Langages validés :"),
128             langageEtudie()
129         ],
130     ),
131 ),
132 )
133
134 ],
135 )
136 )
137 );
138 }
139
140 Column checks() {
141     List<Widget> items = [];
142     listeDeCourses.forEach((course, acheter) {
143         Widget row = Row(
144             mainAxisAlignment: MainAxisAlignment.center,
145             mainAxisAlignment:MainAxisSize.max ,
146             children: [
147                 Text(course),
148                 Checkbox(value: acheter, onChanged: ((newValue){
149                     setState(() {
150                         listeDeCourses[course] = newValue ?? false;
151                     });
152                 })))
153             ],
154         );
155     });

```

```
156     items.add(row);
157
158   });
159   return Column(children: items);
160 }
161
162 Column langageEtudie(){
163   List<Widget> items = [];
164   listeDeCourses.forEach((cours, validate) {
165     print(validate);
166     if (validate == true){
167       Widget row = Row(
168         mainAxisAlignment:MainAxisAlignment.center ,
169         children: [
170           Text(cours)
171         ],
172       );
173       items.add(row);
174     }
175   });
176   return Column(children:items);
177 }
178
179 }
```

Exercice p. 31 Solution n°11

Question 1

Quel widget peut-on utiliser quand le contenu dépasse de l'écran ?

☒ SingleChildScrollView

☐ listView

☐ Scrollview



Le widget « *SingleChildScrollView* » permet de résoudre le problème quand toute l'interface utilisateur ne tient pas sur l'écran. On peut scroller comme une liste.

Question 2

Quel widget permet de faire des séparations entre les widgets ?

☐ Liner

☐ Spacer

☒ Divider




Le widget « *divider* » permet de faire des séparations entre les widgets. On peut l'utiliser pour séparer les onglets dans le menu (*drawer*).

Question 3

Quelle propriété permet de styliser un textField ?


- ☐ textFieldProperty
- ☐ textFieldDecoration
- ☒ decoration

 La propriété « *decoration* » permet de styliser le textField. Elle est associée avec une instance de type « *InputDecoration* ».

Question 4

Quelle propriété permet de définir la fonction lorsqu'on change l'état du widget « *Switch* » ?


- ☐ onSubmit
- ☒ onChanged

 La propriété « *onChanged* » permet de spécifier la fonction exécutée quand on change l'état du bouton « *switch* ».

Question 5

Quel contrôleur peut gérer un textField ?

- ☐ textController
- ☒ TextEditingController
- ☐ ControlerOfTextField

 Le controller « *TextEditingController* » permet de gérer un textField. Il est initialisé dans l'« *initstate* ». Il permet une gestion plus fine du textField.