# Flutter

## Développement Mobile

## Créer un projet Flutter

- Palette de commandes : *Flutter: New Project* > *Empty Application*
  - un nouveau répertoire va être créé à l'endroit spécifié
- On peut aussi utiliser Flutter CLI dans le répertoire souhaité
  - `flutter create -e mon_projet`

## Structure du projet

- Répertoire *lib* : code source
  - *main.dart* : point d'entrée de l'app
- *pubspec.yaml* : configuration et dépendances
- *android*, *ios*, *windows*… : code spécifique à chaque plateforme si besoin
- *web* : code spécifique à la cible web

## Widgets

- En Flutter, tout ce qu'on voit à l'écran est un *widget*
  - *Layouts* : *Grid*, *List*, *Row*, *Column*…
  - *Objects* : *Text*, *Image*, *Icon*, *Button*…
  - l'app elle-même (`MaterialApp`) est un *widget*
- Les *widgets* sont :
  - fournis par le framework
  - ou créés par le développeur à partir de *widgets* existants
  - tout écran que vous créerez sera un *widget*

## *Widgets* sans état

## StatelessWidget

- Un *StatelessWidget* est un *widget* qui ne peut pas être modifié
- Notre page d'accueil sera un *StatelessWidget*
- Snippet *stless* dans VS Code pour créer rapidement un *StatelessWidget*

# Page d'accueil

- Créer fichier *lib/screens/welcome.dart*

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

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

  @override
  Widget build(BuildContext context) {
    return const Placeholder();
  }
}
```

# Adaptation `main.dart`

- `MaterialApp` est un widget qui agit comme conteneur pour l'app
  - permet de configurer l'app (localisation, navigation, thème, etc.)
- *home* est un paramètre obligatoire de `MaterialApp`
  - doit être un *widget*
  - désigne la route par défaut (page d'accueil)
- On va indiquer le chemin vers notre écran d'accueil
  - il faut importer le fichier welcome_screen.dart_ (avec chemin relatif)
  - et instancier le *widget* `WelcomeScreen` dans *home*

```
// main.dart
import 'package:flutter/material.dart';
import 'package:flutter_app/screens/welcome_screen.dart';

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

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

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: WelcomeScreen(),
    );
  }
}
```

# *Widget Scaffold*

- *Scaffold* est un *widget* qui implémente la structure de base d'une app
  - *AppBar*, *Drawer*, *BottomNavigationBar*, *FloatingActionButton*…
- *Scaffold* est souvent utilisé comme racine de l'app

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Center(
        child: Text(
          'Du contenu...',
          style: TextStyle(fontSize: 24),
        ),
      ),
    );
  }
}
```

# *DevTools Widget Inspector*

- *Widget Inspector* est un outil de débogage puissant
  - permet de visualiser la hiérarchie des *widgets* d'une application Flutter
  - il faut s'habituer à l'utiliser, notamment quand on ne comprend pas pourquoi un écran plus complexe ne s'affiche pas comme souhaité
- Pour l'activer sous VS Code :
  - lancer l'app en mode *debug*
  - `F1 > Flutter: Open DevTools > Widget Inspector`

# Les *widgets* de layout

- Les *widgets* de layout permettent de structurer l'interface
- Ils contiennent d'autres *widgets* et définissent comment ils sont affichés
  - par exemple, *Center* centre son contenu horizontalement et verticalement
- Certains possèdent un seul enfant (*child*)
  - ex. : *Center*, *Container*, *Column*, *Row*

- D'autres peuvent en avoir plusieurs (*children*)
  - ex. : *Stack*, *ListView*, *GridView*

## *Widget* - Stack

- *Stack* est un *widget* qui empile ses enfants les uns sur les autres
  - comme un *z-index* en CSS, on empile les *widgets* du fond vers le premier plan
  - les *widgets* de la *Stack* sont spécifiés dans une liste donnée au paramètre *children*
- *Positioned* permet de positionner les *widgets* dans le *Stack*
  - *top*, *bottom*, *left*, *right* pour positionner
  - *width*, *height* pour dimensionner

# Implémentation d'une *Stack*

- On va utiliser une *Stack* pour superposer un texte et un bouton sur une image de fond

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Stack(
        children: [
          Center(
            child: Text(
              'Du contenu...',
              style: TextStyle(fontSize: 24),
            ),
          ),
        ],
      ),
    );
  }
}
```

# Ajout d'une image en fond de Stack

- Plusieurs façons d'inclure des images :
  - *Image.network* : depuis le web
  - *Image.asset* : depuis les ressources du projet

    - *Image.memory* : depuis la mémoire
    - *Image.file* : depuis un fichier local
- On va charger notre image depuis un répertoire dédié du projet
    - on crée un répertoire *assets/images* à la racine du projet

## `pubspec.yaml`

- *pubspec.yaml* est utilisé pour déclarer les dépendances et les ressources du projet
- *pubspec* est un fichier de configuration YAML
    - YAML est un format de sérialisation de données (comme JSON)
    - mais plutôt utilisé pour la configuration
    - l'indentation est impoortante : elle spécifie la hiérarchie
    - puis on a des paires `clé: valeur` pour spécifier la configuration

# Configuration du chemin des images

- Les resources locales sont spécifiées dans la section *flutter* de *pubspec.yaml*
    - on ajoute le chemin de nos images

```
// pubspec.yaml
name: ...

...

flutter:
  uses-material-design: true

  assets:
    - assets/images/
```

# Ajout de l'image sur l'écran d'accueil

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

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

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Accueil')),
      body: const Stack(
        children: [
```

```
          Positioned.fill(
            child: Image.asset(
              'assets/images/sea.jpg',
              fit: BoxFit.cover,
            ),
          ),
          Center(
            child: Text(
              'Du contenu...',
              style: TextStyle(fontSize: 24),
            ),
          ),
        ],
      ),
    );
  }
}
```

# Les boutons

- On va ajouter un bouton à la *Stack*
- Flutter propose plusieurs types de boutons, dont :
  - *TextButton* : bouton simple avec texte
  - *IconButton* : bouton avec icône
  - *OutlinedButton* : bouton avec contour
  - *ElevatedButton* : bouton « rempli »;

# Ajout d'un bouton

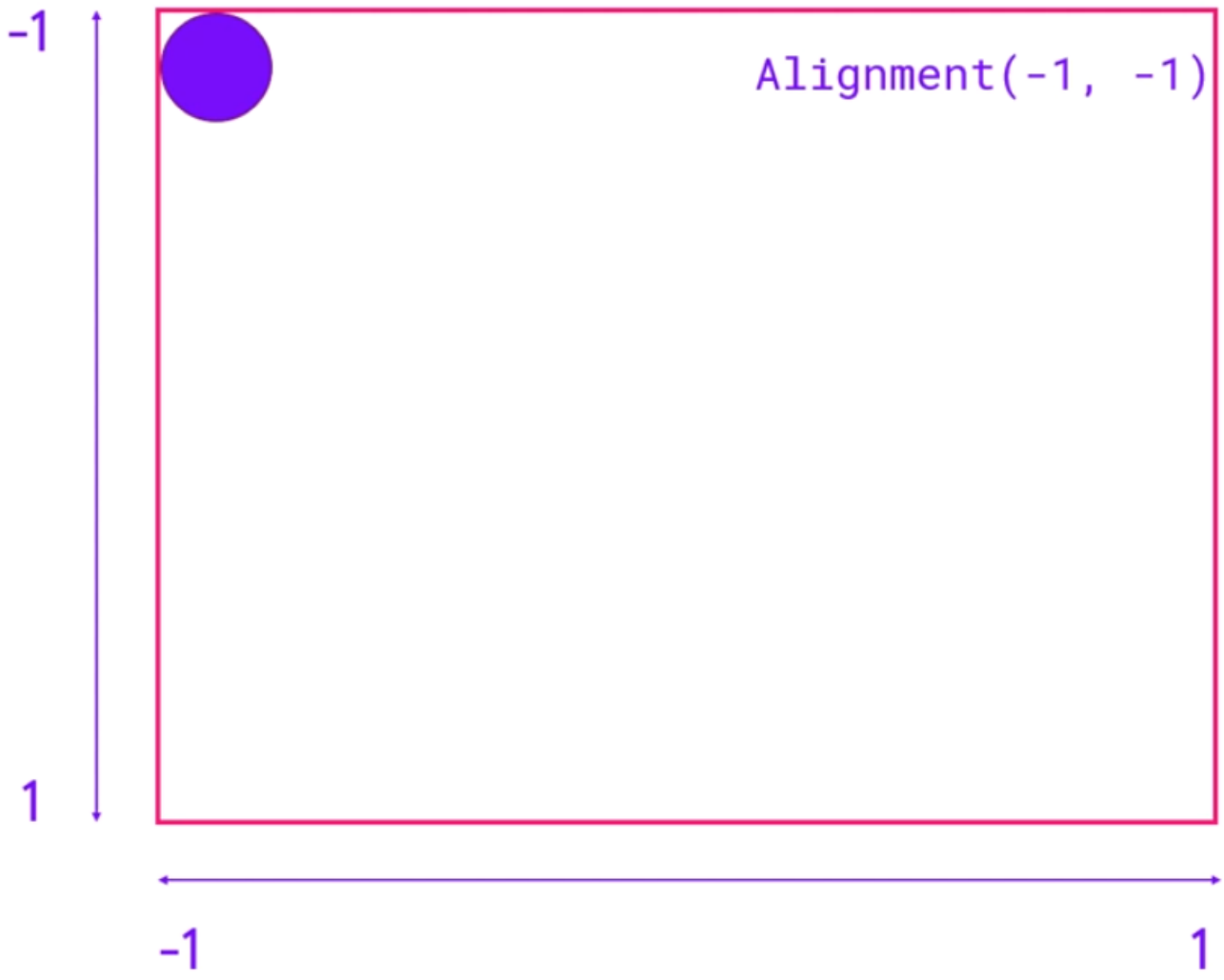
- On va ajouter un *ElevatedButton* en bas de l'écran
- Un *ElevatedButton* requiert :
  - `onPressed` pour définir l'action à effectuer
  - `child` pour définir le contenu du bouton
- On va utiliser le *widget Positioned* pour positionner le bouton
  - les paramètres *top*, *bottom*, *left* et *right* définissent la distance par rapport au bord de l'écran
  - on va « ancrer » le bouton centré en bas, pour cela on utilise juste *bottom*, *left* et *right* avec une valeur identique

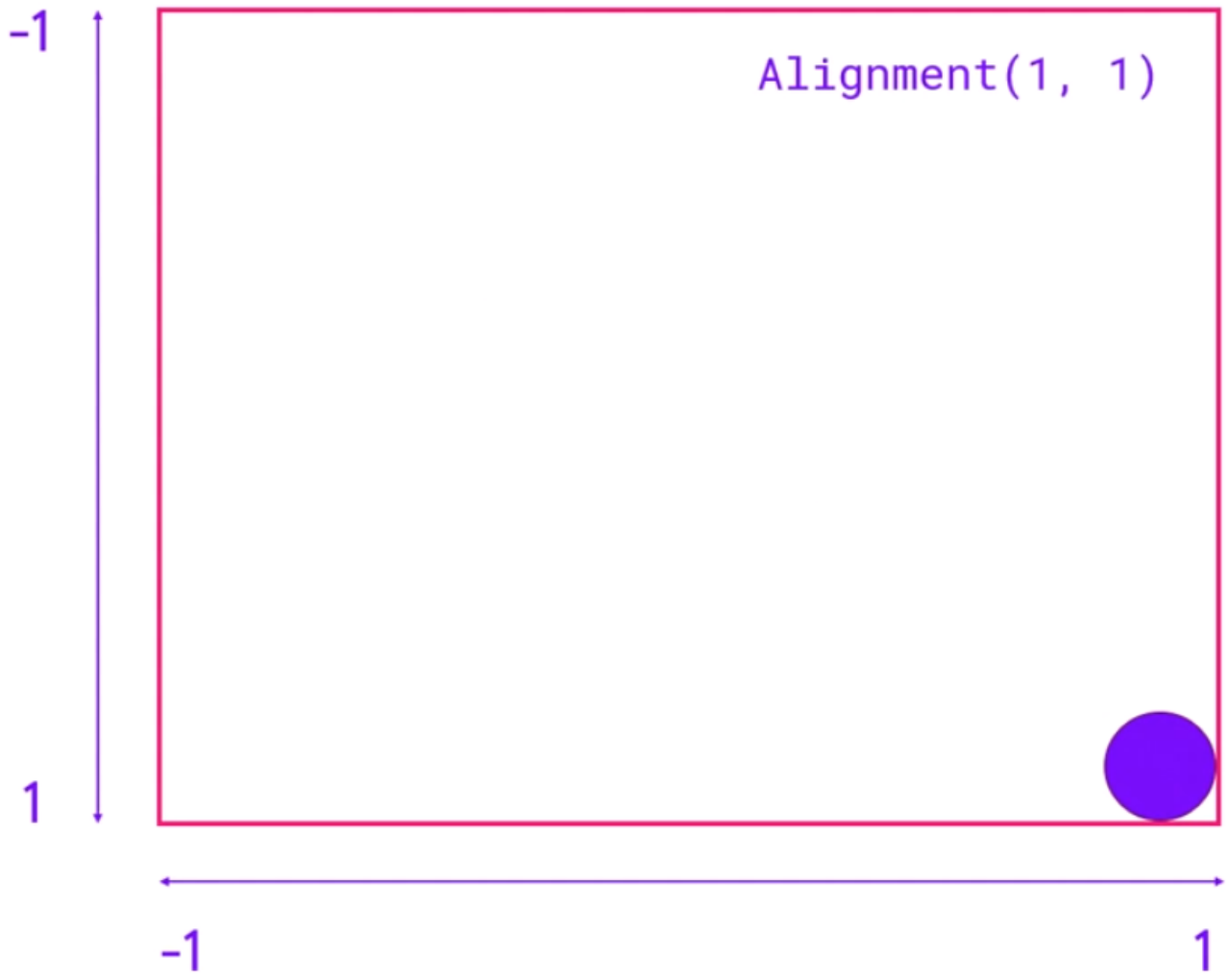
```
...

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Accueil')),
```

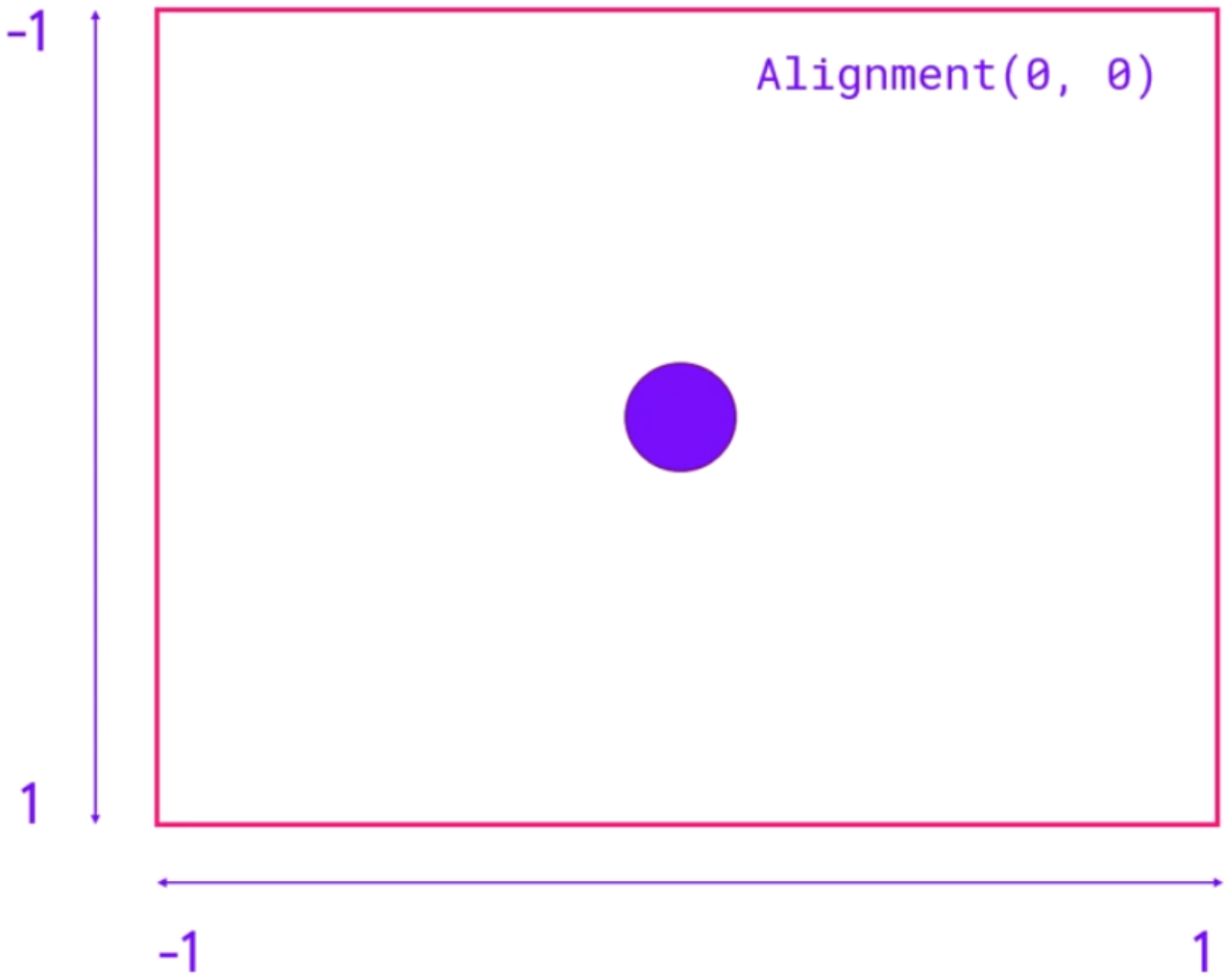
```
    body: Stack(
      children: [
        Positioned.fill(...),  // image
        Center(...),           // texte
        Positioned(            // bouton
          bottom: 16,
          left: 16,
          right: 16,
          child: ElevatedButton(
            onPressed: () {},
            child: const Text('Cliquez-moi'),
          ),
        )
      ],
    ),
  );
}
```

# Positionnement relatif - *Widget* `Align`

- On peut aussi utiliser le *widget Align* pour positionner un *widget* dans le parent
- *Align* prend un paramètre *alignment* et un *child*
  - *alignment* est un objet de type *Alignment*
  - *Alignment* prend deux paramètres : *x* et *y* (valeurs entre -1 et 1)
  - *x* : -1 = tout à gauche, 1 = tout à droite
  - *y* : -1 = tout en haut, 1 = tout en bas


-1
Alignment(-1, -1)
1
-1
1


-1
Alignment(1, 1)
1
-1
1




# Modification du *layout*

- Utilisons des *widgets Align* pour modifier le *layout* de notre *Stack* :
  - texte centré horizontalement et à 25 % de la hauteur
  - bouton centré horizontalement et à 75 % de la hauteur

```
...

Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Accueil')),
    body: Stack(
      children: [
        Positioned.fill(...),  // image
        const Align(
          alignment: Alignment(0, -0.5),
          child: Text(...),   // texte
        ),
        Align(
          alignment: Alignment(0, 0.5),
          child: ElevatedButton(...),  // bouton
```

```
        )
      ],
    ),
  );
}
```

# Améliorations esthétiques

- Modifions le style du texte :
  - couleur blanche (paramètre `color`)
  - ombre portée (paramètre `shadows` - liste)

# Récap

- L'interface est décrite en langage Dart
- Tout ce qu'on voit est un *widget*
  - même la structure de l'UI (le *layout*) est contrôlée par des *widgets*
- La racine de l'app est un *widget* `MaterialApp` (*Material Design*)
- Chaque écran est typiquement structuré par un *widget* `Scaffold`
  - *appBar*, *body*, *bottomNavigationBar*…
- Pour structurer le contenu (*body*), on utilise des *widgets* de *layout*
  - *Column*, *Stack*, *Center Positioned*, *Align*…
  - tous ces *widgets* sont des conteneurs pour un ou plusieurs autres
- Ces autres *widgets* peuvent eux-mêmes être des conteneurs ou bien des objets finaux :
  - *Text*, *Image*, *Icon*, *Button*…
- L'outil *Widget Inspector* permet de visualiser la hiérarchie des *widgets*

# Retour - *Widgets Stateless*

- Dans l'écran créé, on a utilisé des *widgets Stateless*
  - tout ce qui est affiché est statique
- Nous allons maintenant voir comment ajouter de l'interactivité

# *Widgets* avec état

# *Widgets Stateful*

- Un *StatefulWidget* est un *widget* qui peut être modifié
- Nécessaire dès que l'interface peut changer

    - écrire du texte
    - afficher une liste déroulante…
- C'est différent car on doit maintenant gérer un **état** (*state*)
    - quel est le texte affiché ?
    - quelle est la valeur sélectionnée dans la liste déroulante ?
    - que fait-on quand on clique sur le bouton ?

# Quelques exemples de *widgets* avec état

- *StatefulWidget* : gérer l'état de l'interface
- *Navigator* : gérer la navigation entre les écrans
- *TextField* : écrire du texte
- *TextEditingController* : gérer le texte saisi dans un *TextField*
- *DropdownButton* : liste déroulante
- *SharedPreferences* : stocker des données de manière persistante

# Écran « Paramètres »

- On va illustrer l'utilisation des *widgets* décrits précédemment en développant un écran de paramètres pour l'application
    - nom du l'utilisateur (champ texte)
    - choix de l'image de fond (menu déroulant)
    - bouton pour valider les paramètres
    - persistance entre les lancements de l'application

# Implémentation

- Créer un nouvel écran dans un fichier *settings*
    - utiliser un *widget StatefulWidget* (snippet *stful* sous VS Code) nommé *SettingsScreen*
- Différence principale avec *StatelessWidget* : on doit implémenter 2 classes :
    - le *widget* lui-même (*SettingsScreen*)
    - la classe *State* qui gère l'état de l'écran

# *StatelessWidget* vs *StatefulWidget*

- *StatelessWidget* : *widget* qui ne peut pas être modifié
    - classe unique dérive de *StatelessWidget*
    - méthode `build` obligatoire pour décrire l'UI

- *StatefulWidget* : *widget* qui peut être modifié
  - classe dérivée de *StatefulWidget*
    - méthode `createState` obligatoire pour créer une instance de *State*
  - **2ème classe** dérivée de *State* pour gérer l'état
    - méthode `build` obligatoire pour décrire l'UI

# La classe *State*

- La classe *State* décrit l'information :
  - définie à la création du *widget* (**initialisation**)
  - et qui peut varier durant le cycle de vie du *widget* (**modification de l'état**)
- Un *StatefulWidget* reste donc **immuable** (ne change pas), comme un *StatelessWidget*
  - **c'est l'objet *State* associé qui change**

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

class SettingsScreen extends StatefulWidget {
  const SettingsScreen({super.key});

  @override
  State<SettingsScreen> createState() => _SettingsScreenState();
}

class _SettingsScreenState extends State<SettingsScreen> {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Paramètres')),
      body: const Placeholder(),
    );
  }
}
```

# Navigation en Flutter

- Deux façons d'implémenter la navigation :
  - *Navigator* : navigation simple
  - *Router* : navigation plus complexe (gestion des routes, des transitions, etc.)
- On va utiliser *Navigator*

# *Navigator*

- *Navigator* est un *widget* qui gère la navigation entre les écrans
- Fonctionne sur un principe de pile d'écrans :
  - afficher un nouvel écran ? ⇒ on empile un écran sur la pile (*push*)
  - revenir en arrière ? ⇒ on retire un écran de la pile (*pop*)
  - afficher un nouvel écran en supprimant l'actuel ? ⇒ on retire et on empile (*pushAndReplace*)
- *Navigator* est accessible via la méthode `Navigator.of(context)`
  - on appelle ensuite `push`, `pop`… sur cet objet

# Navigation - *push*

- Pour afficher un nouvel écran, on utilise la méthode `push`
- On passe un objet `Route` qui décrit l'écran à afficher
  - spécifiquement, on peut utiliser `MaterialPageRoute` pour afficher un écran avec une transition Material prédéfinie

```
Navigator.of(context).push(
  MaterialPageRoute(
    builder: (BuildContext context) => const SettingsScreen()
  ),
);
```

# Événement déclencheur

- On veut naviguer vers l'écran *SettingsScreen* quand on clique sur le bouton de la page d'accueil
- On doit donc maintenant ajouter un corps à la méthode `onPressed` du *ElevatedButton* (précédemment vide)
  - c'est à cet endroit qu'on va insérer le code de navigation

```
ElevatedButton(
  onPressed: () {
    Navigator.of(context).push(
      MaterialPageRoute(builder: (context) => const SettingsScreen())
    );
  },
  child: const Text('Paramètres'),
),
```

# Implémentation automatique du retour

- Quand on navigue vers un nouvel écran, un bouton « Retour » est automatiquement ajouté dans l'*AppBar*
  - il utilise la méthode `pop` pour revenir à l'écran précédent
- On peut bien sûr personnaliser ce mécanisme

# Persistance des paramètres - le package `shared_preferences`

# Packages

- Les packages sont des composants réutilisables que l'on peut inclure dans nos apps
  - implémentés par les équipes Dart/Flutter
  - ou par la communauté Open Source
- https://pub.dev : repository officiel des packages Dart/Flutter
- On souhaite ajouter un package pour gérer la persistance des paramètres de l'application à travers les lancements
  - une solution est d'utiliser le package `shared_preferences`

# SharedPreferences - Installation

- Trouver le package sur https://pub.dev
- Regarder la doc d'installation (*Installing*)
- Installer localement
- Vérifier que le package est bien ajouté dans *pubspec.yaml*

# SharedPreferences - Présentation

- *SharedPreferences* est une solution de stockage **simple mais limitée**
  - paires clé-valeur
  - nombres (entiers, flottants), booléens, strings, listes de strings
  - ex. : (nom, 'John'), (age, 25), (theme, 'dark')…
- À éviter donc pour :
  - données critiques
  - données volumineuses
  - données complexes (objets, listes complexes…)

# SharedPreferences - Utilisation

```
import 'package:shared_preferences/shared_preferences.dart';

// Récupérer une instance de SharedPreferences
SharedPreferences sp = await SharedPreferences.getInstance();

// Écrire une valeur : clé + valeur
await sp.setInt('age', 25);

// Autres méthodes dispos : setDouble, setBool, setString, setStringList
// Chaque setter a un getter correspondant : getInt, getDouble...

// Lire une valeur
int age = sp.getInt('age');

// Supprimer une valeur
bool estSupprime = await sp.remove('age');
```

# Code asynchrone

- Les exemples de code précédents utilisent le mot-clé `await`
- `await` est utilisé pour attendre le résultat d'une **fonction asynchrone**
  - une **fonction asynchrone** est une fonction qui fait un traitement « long » lié à une opération d'entrée-sortie (disque dur, BDD, réseau...)
- Quand on *await* le résultat d'une fonction asynchrone, on doit se trouver dans une fonction elle-même asynchrone (**marquée par le mot-clé *async***)
- **Une fonction *async* renvoie un objet de type *Future***
  - prend en <paramètre de type> le type de retour de la fonction
  - ex. : `Future<int>`, `Future<String>`, `Future<void>`...
- Retenez le fondamental :
  - *await* toujours avec *async* dans notre code

# Implémentation de la persistance des paramètres

- Fichier *settings_persistence.dart* dans un nouveau répertoire *persistence*
- Classe *SettingsPersistence* avec 2 méthodes :
  - `saveSettings` pour sauvegarder les paramètres
  - `loadSettings` pour charger les paramètres

```
// lib/persistence/settings_persistence.dart
import 'package:shared_preferences/shared_preferences.dart';
```

```
class SettingsPersistence {
  Future<void> saveSettings(String username, String image) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('username', username);
    await prefs.setString('backgroundImage', image);
  }

  Future<Map<String, String>> loadSettings() async {
    final prefs = await SharedPreferences.getInstance();
    final username = prefs.getString('username');
    final backgroundImage = prefs.getString('backgroundImage');
    return {
      'username': username ?? '',
      'backgroundImage': backgroundImage ?? '',
    };
  }
}
```

# Implémentation de l'UI

- Implémentez l'UI de l'écran *SettingsScreen* :
  - champ texte pour le nom
  - liste déroulante pour l'image de fond
  - ces 2 *widgets* seront dans un conteneur *Column*
    - alignement vertical, les uns en dessous des autres
  - bouton en bas à droite pour valider les paramètres

# Implémentez l'UI - TextField

- *TextField* est un *widget* qui permet à l'utilisateur de saisir du texte
- Prend un paramètre `decoration` de type *InputDecoration*
  - paramètre *hintText* = texte affiché quand le champ est vide

```
// Implémentation du TextField
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text('Paramètres')),
    body: const Column(
      children: [
        Text('Nom d\'utilisateur'),
        TextField(
          decoration: InputDecoration(
            hintText: 'Votre nom',
          ),
```

```
        ),
      ],
    ),
  );
}
```

# Implémentation de l'UI - *DropdownButton*

- *DropdownButton* est un *widget* qui affiche une liste déroulante
  - paramètre `value` : valeur sélectionnée
  - paramètre `items` de type *List<DropdownMenuItem>* : les éléments
- chaque élément de la liste est donc un *DropdownMenuItem*
  - paramètre `value` est la valeur de l'élément
  - paramètre `child` est le *widget* à afficher dans la liste déroulante

# *Widget DropdownButton* - Fonctionnement

```
DropdownButton<String>(  // String est le type de la valeur
  value: 'image1',       // la valeur sélectionnée
  items: [               // la liste des éléments
    DropdownMenuItem(    // chaque élément a :
      value: 'image1',        // - une valeur (qui sera consultable)
      child: Text('Image 1'), // - un _widget_ (qui est affiché)
    ),
    DropdownMenuItem(
      value: 'image2',
      child: Text('Image 2'),
    ),
  ],
  onChanged: (value) {
    // Code à exécuter quand on sélectionne une nouvelle valeur
    // La valeur sélectionnée est passée en paramètre ('value')
  },
),
```

```
// Implémentation du DropdownButton de l'écran Paramètres
body: Column(
  children: [
    const Text('Nom d\'utilisateur'),
    const TextField(...),
    const Text('Image de fond'),
    DropdownButton<String>(
      value: 'assets/images/sea.jpg',
      items: const [
        DropdownMenuItem(
```

```
          value: 'assets/images/country.jpg',
          child: Text('Campagne'),
        ),
        DropdownMenuItem(
          value: 'assets/images/lake.jpg',
          child: Text('Lac'),
        ),
        DropdownMenuItem(
          value: 'assets/images/mountain.jpg',
          child: Text('Montagne'),
        ),
        DropdownMenuItem(
          value: 'assets/images/sea.jpg',
          child: Text('Mer'),
        ),
      ],
      onChanged: (value) {},
    ),
  ],
)
```

# Traquer l'élément sélectionné

- Pour conserver la trace de l'élément sélectionné, on va ajouter à notre classe une variable *selectedImage*
  - stockage de la *value* de l'image sélectionnée
  - modifiée dans la méthode `onChanged` du *DropdownButton*

# Mettre à jour l'UI

- Petit problème : si on se contente de mettre à jour la variable *selectedImage* dans la méthode `onChanged`, rien ne se passe à l'écran
  - c'est normal, car l'interface n'est pas mise à jour automatiquement
- **Il faut notifier le framework que l'interface doit changer**
  - pour cela on utilise la méthode `setState`
  - elle indique au framework que l'interface doit être reconstruite
- Une syntaxe permet de passer une fonction anonyme à `setState` pour mettre à jour la variable

```
setState(() {
  // Code qui justifie la mise à jour de l'interface
});
```

```
class _SettingsScreenState extends State<SettingsScreen> {
  var selectedImage = 'assets/images/sea.jpg';
```

```
  @override
  Widget build(BuildContext context) {
      // ...
      DropdownButton<String>(
        value: selectedImage,
        items: const [
          DropdownMenuItem(
            value: 'assets/images/country.jpg',
            child: Text('Campagne'),
          ),
          DropdownMenuItem(
            value: 'assets/images/lake.jpg',
            child: Text('Lac'),
          ),
          DropdownMenuItem(
            value: 'assets/images/mountain.jpg',
            child: Text('Montagne'),
          ),
          DropdownMenuItem(
            value: 'assets/images/sea.jpg',
            child: Text('Mer'),
          ),
        ],
        onChanged: (value) {
          setState(() {
            selectedImage = value ?? 'assets/images/sea.jpg';
          });
        })}}
```

# Lire/écrire dans un *TextField* depuis le code

- On va avoir besoin de manipuler le *TextField* depuis le code
  - initialisation du champ au moment de l'affichage de l'écran
  - récupération de la valeur du champ au moment de la validation
- Une option courante pour lire/écrire dans un *TextField* en code est d'utiliser un ***TextEditingController***
  - on crée une variable *TextEditingController* pour chaque champ de texte (dans la classe *State*)
  - on l'associe au *TextField* via le paramètre `controller` (dans le *build*)
- On peut ensuite manipuler le *TextField* via le la variable créée
  - accès en lecture/écriture via *leController.text*

```
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();

  // ...
```

```
  TextField(
    controller: usernameController,
    decoration: InputDecoration(
    hintText: 'Votre nom'),
  ),
}
```

# Implémentation du bouton de validation

- Le bouton de validation de l'écran de paramètres sera un *FloatingActionButton*
  - un bouton flottant qui apparaît en bas à droite de l'écran
  - on lui associe une icône (*icon*) et une action (*onPressed*)
- Bonne pratique : extraire la logique de sauvegarde des paramètres dans une méthode à part
- Le code associé utilisera la classe *SettingsPersistence* pour sauvegarder les paramètres

# Icônes Material

- Le thème *Material* propose un grand nombre d'icônes prédéfinies
  - https://api.flutter.dev/flutter/material/Icons-class.html
- Listées par catégories sur le site officiel de *Material Design*
  - https://material.io/icons/
- Bien sûr, vous pouvez utiliser d'autres icônes

```
// Implémentation du bouton de validation
FloatingActionButton(
  child: const Icon(Icons.save),
  onPressed: () async {
    saveSettings();  // méthode séparée, à écrire dans la classe
  }
)
```

# Méthode `saveSettings`

- On va ajouter la méthode `saveSettings` à la classe *SettingsScreenState*
  - cette méthode va récupérer les valeurs des *TextFields* et du *DropdownButton*
  - puis les sauvegarder dans les *SharedPreferences*

```
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();
  var selectedImage = 'assets/images/sea.jpg';
```

```
  // ...

  Future<void> saveSettings() async {
    final username = usernameController.text;
    final image = selectedImage;

    final sp = SettingsPersistence();
    await sp.saveSettings(username, image);
  }
}
```

# Initialisation de l'écran Paramètres

- Lorsque l'écran Paramètres est affiché, il faut que les paramètres enregistrés soient chargés
- Pour cela, on va écrire une méthode *loadSettings* qui va de nouveau utiliser la classe *SettingsPersistence*
- Cette méthode doit être appelée à chaque fois que l'écran est affiché
- On a besoin d'un mécanisme pour exécuter du code au moment du chargement de l'écran
  - méthode prédéfinie *initState*, déclenchée automatiquement au chargement de l'écran

```
class _SettingsScreenState extends State<SettingsScreen> {
  final usernameController = TextEditingController();
  var selectedImage = 'assets/images/sea.jpg';

  // initState : appelée automatiquement au chargement de l'écran
  @override
  void initState() {
    super.initState(); // appel à la méthode parente, obligatoire
    loadSettings();    // chargement des paramètres
  }

  // ...

  // Méthode de chargement des paramètres
  // Utilise la classe SettingsPersistence
  Future<void> loadSettings() async {
    final sp = SettingsPersistence();
    final settings = await sp.loadSettings();
    setState(() {
      usernameController.text = settings['username'] ?? '';
      selectedImage = settings['backgroundImage'] ?? 'assets/images/sea.jpg';
    });
  }
}
```

# Confirmation de l'enregistrement

- Actuellement, aucun retour (*feedback*) n'est donné à l'utilisateur après l'enregistrement des paramètres
  - Mauvaise UX (expérience utilisateur, *User eXperience*)
- On va ajouter un *SnackBar* pour afficher un message de confirmation
  - un *SnackBar* est un message éphémère qui apparaît en bas de l'écran
- On va utiliser la méthode `showSnackBar` de l'objet *ScaffoldMessenger*
  - on lui passe un *SnackBar* avec le message à afficher
- Il faut appeler `showSnackBar` après la sauvegarde des paramètres dans la propriété `onPressed` du bouton de validation

```
floatingActionButton: FloatingActionButton(
  child: const Icon(Icons.save),
  onPressed: () {
    saveSettings();
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(
        content: Text('Paramètres sauvegardés'),
        duration: Duration(seconds: 3),
      ),
    );
  },
),
```

# Amélioration visuelle - *Widget Padding*

- Comme en CSS, on peut ajouter des marges autour d'un *widget*
- En Flutter, on utilise le *widget Padding*
  - prend un paramètre `padding` de type *EdgeInsets*
  - *EdgeInsets* est un objet qui définit les marges
- On va envelopper le *widget Column* avec un *Padding* pour ajouter de l'espace autour de notre interface

```
// Ajout de marges autour de l'interface
body: Padding(
  padding: const EdgeInsets.all(16),
  child: Column(
      // ...
    ],
  ),
),
```

# Utilisation des paramètres dans l'écran d'accueil

- Deux fonctionnalités à implémenter :
  - afficher « Bienvenue … ! » avec le nom de l'utilisateur dans l'écran d'accueil
  - utiliser l'image de fond choisie

# Implémentation

- Problème : on ne peut pas redéfinir *initState* dans la classe *WelcomeScreen*
  - *StalelessWidget* ⇒ pas d'état !
- Solution : convertir le *widget* en *StatefulWidget*
- 2 variables d'instances pour « retenir » le nom et l'image
  - initialisées au chargement (*initState* délègue à une nouvelle méthode *loadSettings*)
  - utilisées dans le *build* pour afficher le nom et l'image

```
class WelcomeScreen extends StatefulWidget {
  const WelcomeScreen({super.key});

  @override
  State<WelcomeScreen> createState() => _WelcomeScreenState();
}

class _WelcomeScreenState extends State<WelcomeScreen> {
  String username = '';
  String backgroundImage = 'assets/images/sea.jpg';

  // Redéfinition de initState pour appeler loadSettings
  @override
  void initState() {
    super.initState();
    loadSettings();
  }

  @override
  Widget build(BuildContext context) {
    // ...
    child: Image.asset(
      backgroundImage,  // la variable à la place du chemin en dur
      fit: BoxFit.cover,
    ),
    // ...
    child: Text(
      'Bienvenue, $username !', // idem, ici en $interpolation
    // ...
    )
  }
```

```
  // Méthode de chargement des paramètres
  // Similaire à celle de SettingsScreen
  void loadSettings() async {
    final settings = await SettingsPersistence().loadSettings();
    setState(() {
      username = settings['username'] ?? '';
      backgroundImage = settings['backgroundImage'] ?? 'assets/images/sea.jpg';
    });
  }
}
```

# La méthode *dispose*

- La méthode *dispose* d'un *State* est appelée juste avant la destruction du *widget*
  - c'est l'endroit où l'on doit libérer des ressources
  - les ressources non libérées peuvent causer des **fuites mémoire**
- Dans *SettingsScreen*, on a créé un *TextEditingController*
  - c'est une ressource car il est associé à un *listener* qui « écoute » les changements du *TextField*
  - lorsque le *widget* est détruit, la variable *controller* n'est plus accessible par votre code mais reste en mémoire car un *listener* y fait toujours référence !

# *dispose* : quelles ressources ?

- Voici le type de ressources qu'il faut typiquement libérer dans la méthode *dispose* :
  - **animations**
  - **streams** (fichiers…)
  - **controllers** (plusieurs catégories de controllers existent)
  - **sockets** (réseau)

# *dispose* : implémentation

```
class _SettingsScreenState extends State<SettingsScreen> {
  @override
  void dispose() {
    usernameController.dispose(); // libération du controller du TextField
    super.dispose();     // toujours appeler la méthode parente en dernier
  }
}
```

# En résumé (1)

- *Stateless* : on implémente la méthode `build` pour décrire l'UI
- *Stateful* : 2 classes, la « viande » est plutôt dans la classe *State*
  - on a des variables d'instance pour les valeurs dont on veut contrôler l'état
  - on redéfinit :
    - `initState` pour initialiser l'état
    - `build` pour décrire l'UI
    - `dispose` pour libérer les ressources
  - on appelle `setState` chaque fois que l'on doit modifier l'UI

# En résumé (2)

- *Navigator* pour la navigation entre les écrans
  - `push`, `pop`, `pushAndReplace`
- packages
  - *SharedPreferences* pour la persistance des paramètres
  - pub.dev
  - `flutter pub add···` ⇒ *pubspec.yaml*
- Programmation asynchrone
  - *await*, *async*, *Future*, *then*

# Références

- Doc officielle : https://flutter.dev/docs
- Tous les *widgets*, avec des exemples de code : https://flutter.dev/docs/development/ui/widgets
- Communauté Flutter : https://flutter.dev/community
- Chaîne YT *Google Devs Flutter* : https://www.youtube.com/c/flutterdev
- Thème *Material* : https://material.io
- Composants *Cupertino* : https://flutter.dev/docs/development/ui/widgets/cupertino