PROGRAMMATION MOBILE ANDROID

TP 1

STOCKAGE

charles.meunier@u-bourgogne.fr

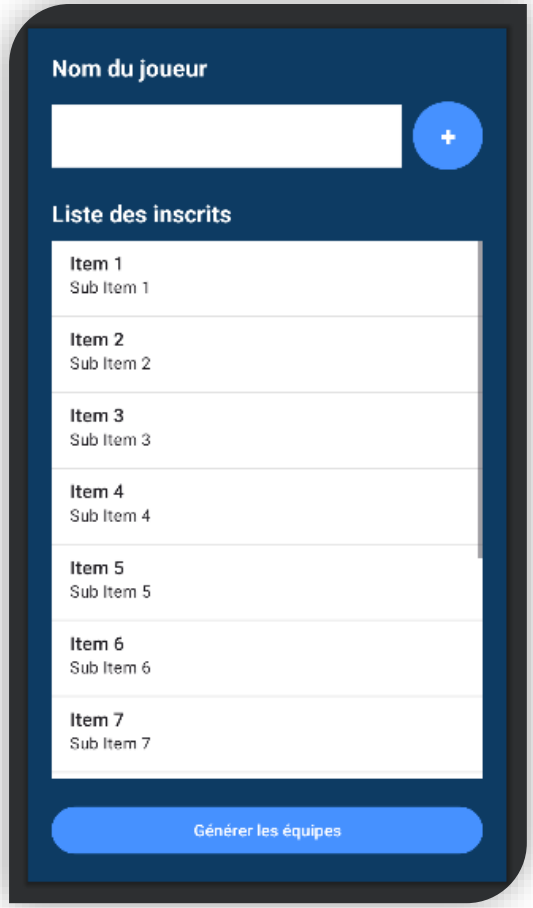
OBJECTIFS

- Alimenter et actualiser une ListView
- Stocker des informations localement
- Découvrir les coroutines Kotlin

EXERCICE 1.1 - INTERFACE

Le président de l'ASQ Badminton, Monsieur Barthélémy Heyrman, organise un tournoi interne à l'occasion de la galette des Rois. Il souhaite disposer d'une application lui permettant de saisir le nom des participants pour ensuite générer aléatoirement des équipes.

Vous allez concevoir cette application !



Nouveau projet

- Créez un nouveau projet selon la méthode vue lors du TD 1.

Interface

- Reproduisez l'interface ci-contre :

  Important : ne perdez pas de temps sur la mise en forme. Concentrez vous sur la suite du TP. Vous peaufinerez plus tard, s'il vous reste un peu de temps.

EXERCICE 1.2 - SAISIE DES NOMS DES PARTICIPANTS

Liste des joueurs

- Ajoutez à `MainActivity` un attribut `players` de type `ArrayList<String>`

- Ajoutez une méthode `addPlayerName` qui ajoutera à la liste `players` le nom du joueur qui aura été saisi dans le champ texte.
  *Rappel : respectez bien la signature suivante pour la fonction **`addPlayerName`** afin de pouvoir l'affecter à l'événement **`onClick`** du bouton **`+`***

```
public fun addPlayerName(view: View)
{

}
```

- Placez un point d'arrêt dans la méthode `addPlayerName` et vérifiez en mode débogage que les noms saisis sont bien ajoutés à l'`ArrayList`.

EXERCICE 1.3 - AFFICHAGE DES NOMS

Layout

- Ajoutez un layout `player_list_item` à votre projet et reproduisez l'interface ci-dessous. Ce layout sera utilisé pour afficher les éléments de la liste des joueurs.

Adaptateur

- Comme vu dans le TD 2, ajoutez à votre projet une classe `PlayerAdapter` qui implémente de `BaseAdpter` et redéfinissez les méthodes `getItemId`, `getItem`, `getCount` et `getView` de l'interface `BaseAdpter`.

- Ajoutez à `MainActivity` un attribut `adapter` de type `PlayerAdapter` et initialisez le dans la méthode `onCreate`.

Configuration de la ListView

- Ajoutez à `MainActivity` une méthode `initializePlayersList` qui affectera l'adaptateur précédemment créé à la ListView de votre interface.

Actualisation de la ListView

- Dans la méthode `addPlayerName`, appelez la méthode `notifyDataSetChanged` de l'adaptateur afin d'actualiser la ListView à chaque ajout d'un nouveau nom.

- Testez le bon fonctionnement

EXERCICE 2.1 - MÉMORISATION DES DONNÉES

PlayerStorage

- Ajoutez à votre projet une classe `PlayerStorage`
- Définissez les méthodes suivantes pour la classe `PlayerStorage` et laissez les vides pour le moment :
  - `constructor`
  - `write`
  - `read`

Context

- Ajoutez un attribut `context` de type `Context` à la classe `PlayerStorage`.
- Modifiez la méthode `constructor` afin qu'elle initialise l'attribut précédent à l'aide d'un `Context` qui lui sera passé en paramètre.

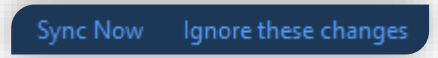
EXERCICE 2.2 - DATASTORE

Ajout d'une dépendance

- Ouvrez le fichier `Gradle Scripts/build.gradle.kts (Module :app)`
- Dans la section `dependencies`, ajoutez la ligne suivante :

```
implementation("androidx.datastore:datastore-preferences:1.0.0")
```

- Puis cliquez sur `Sync Now` dans la barre bleue qui est apparue en haut de votre écran :



Création du datastore

- Dans le fichier `PlayerStorage.kt`, avant la déclaration de la classe `PlayerStorage`, ajoutez la ligne suivante :

```
private val Context.playersStore by preferencesDataStore(name = "players");

class PlayerStorage {
```

*Note : le code précédent ajoute un attribut privé **`playersStore`** à la classe **`Context`**. De cette façon, toute instance de la classe **`Context`** disposera d'un attribut **`playersStore`**.*

*Par ailleurs, le mot clé by permet de déléguer la propriété **`playersStore`** à une autre classe qui gérera la logique liée au stockage de données. Ici, il s'agit d'un fichier de préférences stockant les informations sous la forme de paires clé/valeur.*

*Cette notation est utilisée afin de s'assurer que le DataStore **`players`** n'est créé qu'une seule fois dans toute la durée du programme.*

EXERCICE 2.3 - ECRITURE DES DONNÉES

Clé

Comme vu précédemment, les données sont stockées par sous forme de paires clé/valeur. Mais la clé n'est pas une simple chaine de caractères. Il s'agit d'un objet contenant le nom de la clé mais également le type de données qui lui est associé.

- Ajoutez à la classe `PlayerStorage` un attribut `playersKey` :

```
private var playersKey = stringPreferencesKey("players");
```

*Note : la fonction **`stringPreferencesKey`** retourne un objet de type **`Preferences.Key`** dont la propriété **`name`** est initialisé à **`players`** et dont le type de données associé est **`String`**.*

Enregistrement

- Modifiez la fonction `write` pour qu'elle prenne en paramètre la liste des noms de joueurs.
- Dans la fonction `write`, fusionnez la liste des noms en une chaine de caractères constituez des noms de joueurs séparés par une virgule.
- Enregistrez cette chaine de caractères grâce au code ci-contre :

```
suspend fun write(players: ArrayList<String>)
{
    TODO("Fusionner les éléments de la liste")

    this.context.playersStore.edit { preferences ->
        preferences[playersKey] = data;
    }
}
```

L'édition d'un DataStore est une opération qui doit être définie comme asynchrone de manière à ne pas bloquer les autres tâches le temps de son exécution.

- Ajoutez le mot clé `suspend` au début de la déclaration de la fonction `write`.

EXERCICE 2.4 - LECTURE DES DONNÉES

Lecture

- Modifiez la fonction `read` pour qu'elle retourne une liste de `String` qui correspondra à la liste des noms des joueurs.
- Récupérez les données sauvegardées à l'aide du code suivant :

```
val data = context.playersStore.data.firstOrNull()?.get(playersKey);
```

- Décomposez la chaine de caractères obtenue (si elle existe) en utilisant la virgule comme séparateur.
- Retournez le tableau de `String` ainsi obtenu.

Comme pour l'édition, La lecture d'un DataStore est une opération qui doit être définie comme asynchrone.

- Ajoutez le mot clé `suspend` au début de la déclaration de la fonction `read`.

EXERCICE 2.5 - COROUTINE

Kotlin intègre un outil permettant de gérer facilement l'aspect multitâches des applications sans avoir forcément recours au threads : les `coroutines`.

Plusieurs coroutines peuvent s'exécuter au sein d'un même thread sans bloquer les autres tâches en cours d'exécution. Les coroutines permettent ainsi d'exécuter des tâches asynchrones, c'est à dire qui s'exécutent sans bloquer les autres.

Les coroutines doivent s'exécuter au sein d'une portée spécifique (`scope`) qui permet d'appeler des méthodes asynchrones au sein d'un code qui s'exécute séquentiellement.

Scope

- Ajoutez à la classe `MainActivity` un attribut `mainScope` initialisé avec la méthode `MainScope()`.

  Le scope ainsi créé sera utilisé plus tard pour l'enregistrement et la lecture des données.

```
private val mainScope = MainScope();
```

EXERCICE 2.6 - MÉMORISATION DES JOUEURS

Enregistrement

- Ajoutez à la classe `MainActivity` une méthode `savePlayersList` qui instanciera un `PlayerStorage` et appellera sa méthode `write` dans le scope défini précédemment :

- Appelez `savePlayersList` dans la méthode `addPlayerName` afin d'enregistrer la liste des joueurs après avoir ajouter un participant.

Chargement

- Ajoutez à la classe `MainActivity` une méthode `loadPlayersList` qui instanciera un `PlayerStorage` et appellera sa méthode `read` dans le scope défini précédemment.

- Redéfinissez la méthode `onResume` de `MainActivity` et appelez la méthode `loadPlayersList`.

Tests

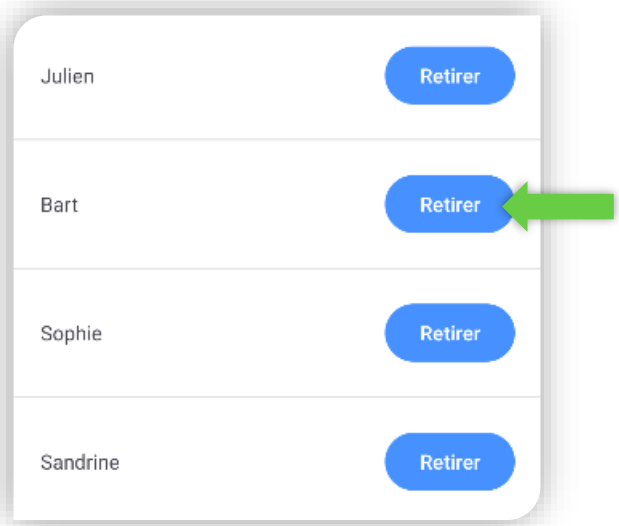
- Testez le bon fonctionnement en ajoutant des joueurs et en relançant l'application.

```
mainScope.launch {
    // Coroutines à exécuter
};
```

BONUS

Suppression des joueur

- Faites en sorte que lorsque l'on clique sur le bouton Retirer d'un joueur, son nom soit supprimer de la liste.



Liste des équipes

- Ajoutez une activité qui affichera la liste des équipes générées aléatoirement. A chaque ouverture de l'activité, les équipes sont regénérées.

- Un bouton Retour permet de revenir à la liste des joueurs.