Haute École Bruxelles-Brabant
École Supérieure d'Informatique
Bachelor en Informatique

2021 – 2022
MOBG5
JLC - SDR

MOBG5 – Développement mobile

Structurer son application

Cycle de vie, MVVM et base de données locales

Consignes

Tous les exercices proposés dans ce TD sont disponibles sur `https://codelabs.developers.google.com/android-kotlin-fundamentals/`.
Nous ne ferons pas tous les exercices de ce tutoriel mais seulement une sélection d'exercices essentiels pour développer une première application.
Lisez les explications présentes dans ce TD et réalisez les exercices en ligne.
Les challenges et devoirs (Homeworks) proposés en ligne sont optionnels.

1 Application `DessertClicker`

Cycle de vie : Cas général

Dans cet exercice vous allez explorer l'utilisation des méthodes `onCreate()`, `onStart()`, `onResume()`, `onPause()`,...

Jusqu'à présent dans chaque activité, vous avez dû réécrire une méthode `onCreate()` qui ressemble à

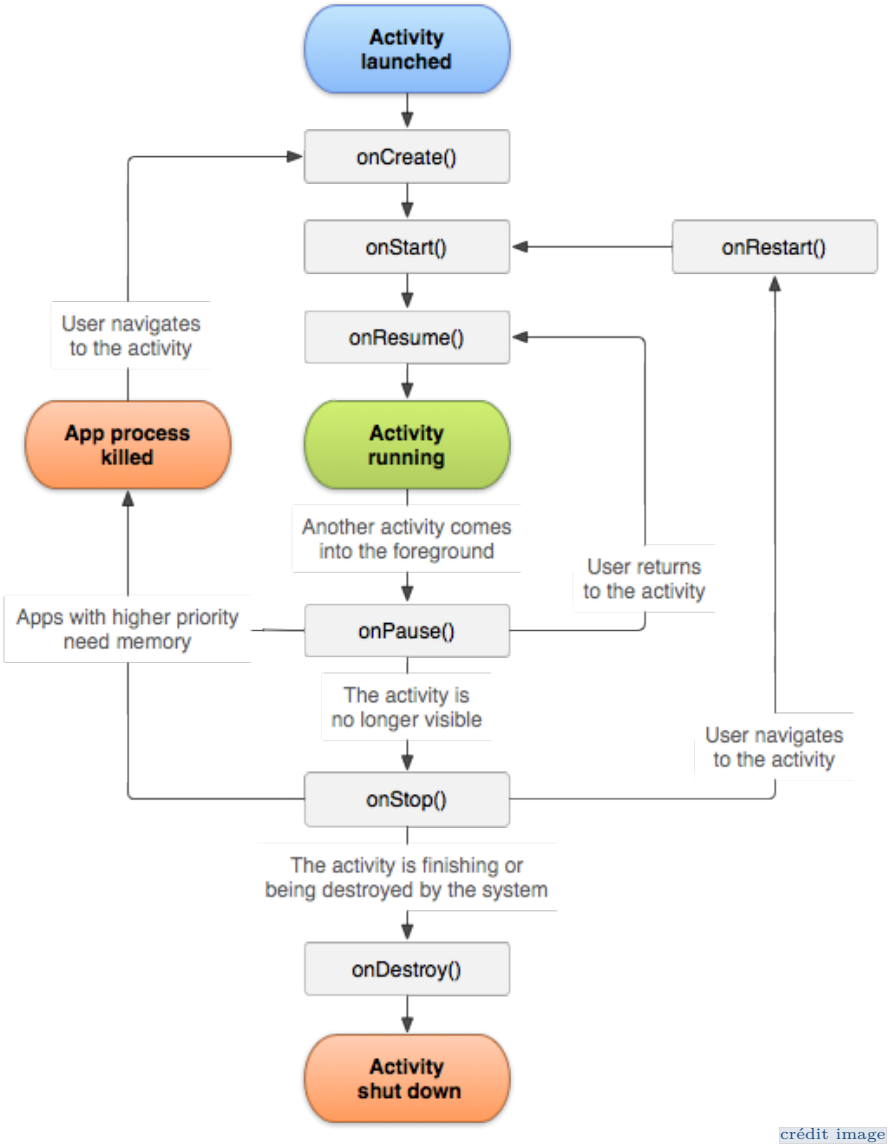
```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    diceImage = findViewById(R.id.dice_image)

    val rollButton: Button = findViewById(R.id.roll_button)
    rollButton.setOnClickListener { rollDice() }
}
```

Cette méthode est appelée automatiquement par `Android` lorsque l'activité est créée.

> Une application Android suis un **cycle de vie** bien défini. En d'autres mots l'application va passer d'état en état en fonction de différentes interactions. A chaque transition entre états une méthode de l'activité est appelée (parfois plusieurs méthodes sont appelées successivement).

Ce cycle de vie de l' activité est décris par la figure ci-dessous.

Version du document
8 septembre 2021



crédit image

Vous trouverez toute les informations sur ce cycle dans la documentation, mais on peut retenir quatre états fondamentaux :

- ▷ l'activité est active, elle occupe le premier plan de l'écran ;
- ▷ l'activité n'est plus au premier plan mais est toujours visible, comme lorsque vous recevez un appel, ou un message ;
- ▷ l'activité n'occupe plus l'écran, elle est cachée mais son état est conservé en mémoire ;
- ▷ `Android` peut demander de terminer une activité et de la retirer de la mémoire, son état devra être sauvegardé afin de pouvoir la redémarrer.

Le changement d'état d'une activité lancera une méthode qualifiée de **CallBack** : `onCreate()`, `onStart()`, `onResume()`, `onPause()`,...

Notons que les fragments comme les activités possèdent un cycle de vie.

Afin de comprendre ces différents cycle de vie, vous allez télécharger l'application `DessertClicker` et la modifier en y ajoutant des messages qui apparaîtront dans la console.

Lien vers l'exercice

04.1 : Lifecycles and logging [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-lifecycles-logging/index.html

Cycle de vie : Cas particuliers

Le principe du cycle de vie compris, il faut en détailler certains cas particuliers.

Tout ce qui a un début a une fin

Si un processus (un `Thread` par exemple) est démarré au début d'une activité via la méthode `onStart()`, ce processus doit être arrêté via la méthode `onStop()`. Dans le cas contraire des incohérences peuvent apparaître au sein de votre application lorsque vous mettez votre application en pause.

Au lieu d'ajouter des instructions au sein des méthodes `onStart()` et `onStop()` de l'activité, une autre approche est de créer un processus (un `Thread`) qui hérite de la classe `LifecycleObserver`.

Une classe qui hérite de `LifecycleObserver` peut observer le cycle de vie de l'activé pour suivre son évolution et se synchroniser avec celle-ci.

Perte de données

Si une activité en pause, c'est à dire non visible à l'écran, est détruite par Android, les données en mémoire sont perdues. Le redémarrage de l'activité peut être compromis.

La méthode `onSaveInstanceState()` d'une activité permet de sauvegarder les données nécessaires au redémarrage de l'activité sous la forme d'un `Bundle`.

Un `Bundle` est une structure clé-valeur disposant de différentes méthodes pour gérer son contenu.

```
bundle = Bundle()
bundle.putString("username", "SpongeBob")
bundle.putBoolean("prof", true)

bundle.getString("username", "Default Value")
bundle.getBoolean("prof", false)
```

Le but de l'exercice étant de comprendre comment gérer la destruction d'une activité, vous allez utiliser l'outil en ligne de commande `Android Debug Bridge`.

adb permet de se connecter à votre téléphone ou à votre émulateur et met à disposition une console Unix pour exécuter différentes commandes.

Par exemple grâce à `adb` on peut installer une application, déposer des fichiers sur l'appareil ou encore appeler le gestionnaire d'activité afin de demander la destruction d'une activité.

Tourne, tourne, tourne

Le troisième cas étudié est celui de la rotation de l'appareil.

> Lors du passage d'un appareil du mode portrait vers le mode paysage, l'activité liée au mode portrait est détruite puis une activité liée au mode paysage est créée..

Lien vers l'exercice

04.2 : Complex lifecycle situations [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-complex-lifecycle/index.html

2 Application `GuessTheWord`

Le ViewModel

Afin de donner une structure aux applications développées, `Android` propose d'utiliser l'architecture **Modèle-Vue-Vue modèle**.

Cette architecture a pour principales caractéristiques de lier la vue et les données affichées via une classe appelées `Vue modèle` et de synchroniser les données affichées via un système de `Binding`. Rappelons que le système de `Binding` permet de rendre le code de la mise à jour de la vue plus lisible.

> Le grand avantage de la classe `Vue modèle` proposé par Android est qu'elle est résistante aux changements de cycle de vie.

Cette résilience signifie qu'on peut conserver dans la classe `Vue modèle` des données en cas de changement de configuration, comme lors des rotations de l'appareil. On peut également échanger des données entre fragments en conservant ces données au sein du `Vue modèle`.

Lien vers l'exercice

05.1 : ViewModel [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-view-model

Des attributs observés : `LiveData`

Dans l'architecture **Modèle-Vue-Vue modèle** certains attributs du `Vue modèle` sont synchronisés avec les données affichées de la vue. Autrement dit, dès qu'un changement

à lieu sur l'un, il est répercuté sur l'autre.

L'utilisation d'attributs implémentant l'`Observateur-Observé` est pertinent pour résoudre ce problème. Cependant nous souhaiterions que les notifications envoyées pour mettre à jour la vue soient envoyées uniquement lorsque la vue est active. C'est pourquoi il faut un `Observateur-Observé` sensible au cycle de vie.

La classe `LiveData` est un **conteneur** d'objet **observable** qui réagit au **cycle de vie** d'un autre composant (activité ou fragment par exemple).

Dans le prochain exercice vous utiliserez une propriété `notification`. Cette propriété est un `LiveData` immuable qui contient un booléen.

```
val notification: LiveData<Boolean>()
```

Si on souhaite transformer ce `LiveData` en un objet variable, on peut écrire :

```
val notification: MutableLiveData<Boolean>()
```

Backing property et encapsulation

Rappelons que l'utilisation des `Backing Properties` peut servir a encapsuler des propriétés dans une classe. Par exemple on peut :

- ▷ créer une propriété privée avec une liste variable ;
- ▷ créer une propriété public immuable dont le champs pointe vers la propriété privée.

```
private val _users = mutableListOf<User>()
val users: List<User>
   get() = _users
```

A l'intérieur de la classe, on peut mettre à jour la liste d'utilisateurs mais à l'extérieur de la classe on a uniquement accès à l'accesseur qui retourne une liste immuable.

L'exercice ci-dessous va mettre en pratique ce principe avec des `LiveData` à la place des listes.

Lien vers l'exercice

05.2 : LiveData and LiveData observers [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-live-data

Retour sur le Data Binding

Pour l'instant les vues utilisent les attributs et méthodes disponibles dans l'activité ou le fragment associé. Les méthodes utilisées appelent alors une méthode dans le `Vue modèle`.

L'objectif est de supprimer l'intermédiaire, il faut transformer la vue pour qu'elle accède directement au `Vue modèle`. De cette façon du code inutile dans l'activité ou le fragment peut être supprimé.

L'attribut `binding` utilisé jusqu'à présent dans l'activité a dorénavant une référence au `Vue modèle`.

Lien vers l'exercice

05.3 : Data binding with ViewModel and LiveData [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-live-data-data-binding

Formatage des attributs affichés

La vue est simplifiée et contient les références aux attributs à afficher en provenance du `Vue modèle`. Cependant lorsque cet attribut doit être formaté (comme une date ou une heure), une transformation doit être apportée à la donnée. Pour ce faire vous allez utiliser la classe `Transformations` afin de formater les valeurs retournées par les accesseurs des propriétés du `Vue modèle` .

Lien vers l'exercice

05.4 : LiveData transformations [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-live-data-transformations

3 Application `TrackMySleepQuality`

Créer une base de données locale

Chaque application peut conserver ses données sur l'appareil via une base de données `SQLite`. Afin d'accéder à cette base de données, une surcouche intitulée `Room` permet à l'application d'effectuer des requêtes dans cette base de données.

Afin que la connexion à cette base de données soit conservée lors d'un changement de configuration, comme une rotation de l'appareil, l'instance de `Room` est conservée dans le `Vue modèle`.

Les objets représentants les tables de votre base de données sont appelés des `Entités` et les requêtes sont écrites dans des classes `Data Access Object`. Heureusement les requêtes classiques sont générées automatiquement. par exemple pour insérer un élément dans une table d'utilisateurs, on écrit :

```
@Insert
fun insert(user: UserEntity)
```

Kotlin companion object

Rappelons que si une méthode ou une propriété est liée à une classe et non à une instance de classe, on peut utiliser le mots clés `companion object`.

Par exemple on peut créer un `companion` de la manière suivante :

```
class Cafe {
   companion object {
      const val LATTE = "latte"
   } fun bestBeverage() = LATTE
}
```

Pour utiliser ce `companion`, il suffit d'utiliser le nom de la classe et non son instance :

```
Cafe.LATTE
```

La similitude avec la notion de `static` en Java est évidente.

Lien vers l'exercice

06.1 : Create a Room database [a]

a. https://codelabs.developers.google.com/codelabs/kotlin-android-training-room-database