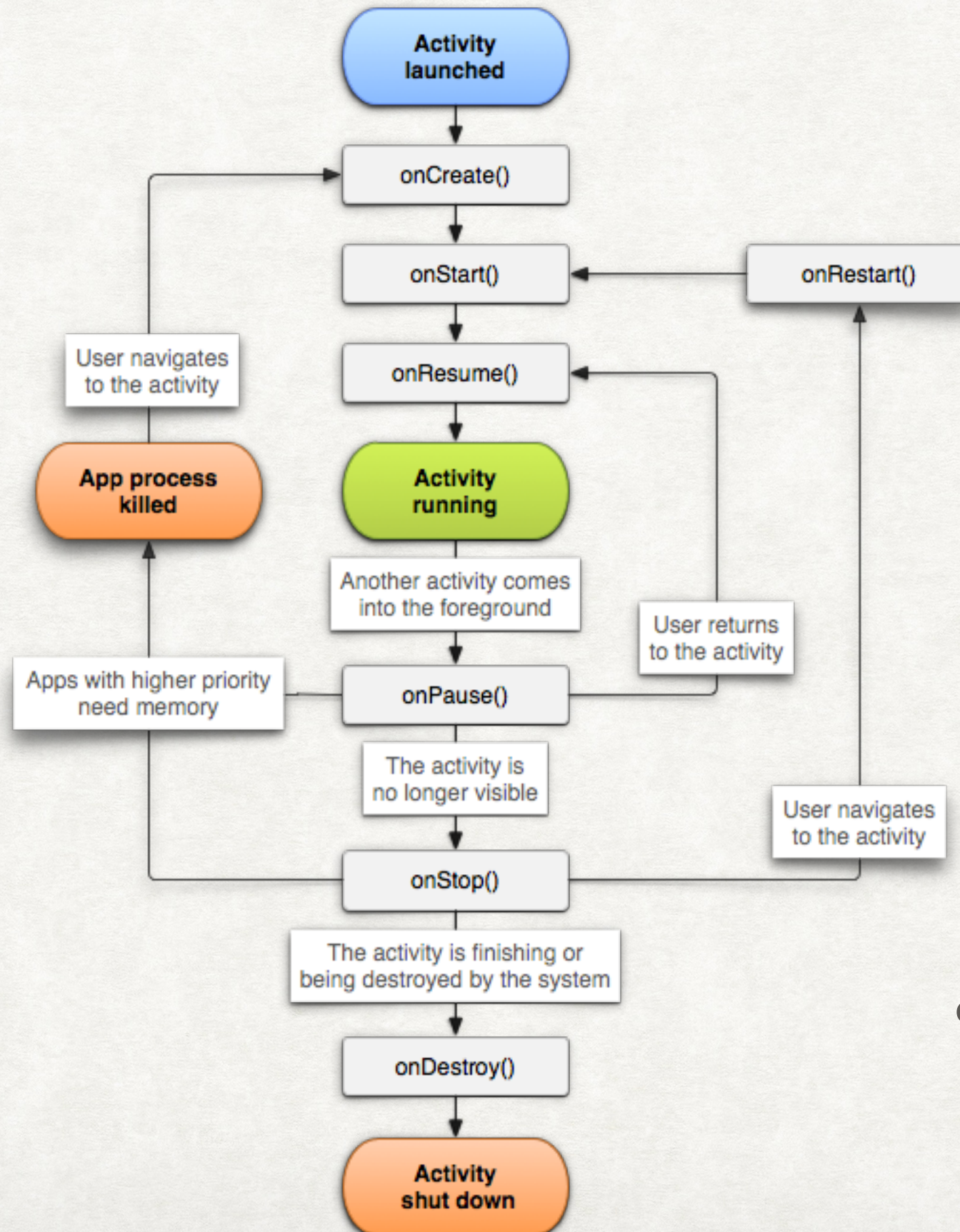


PROGRAMMATION DE COMPOSANTS MOBILES (ANDROID)

ViewModel et LiveData

WIESLAW ZIELONKA

Cycle de vie de l'activité



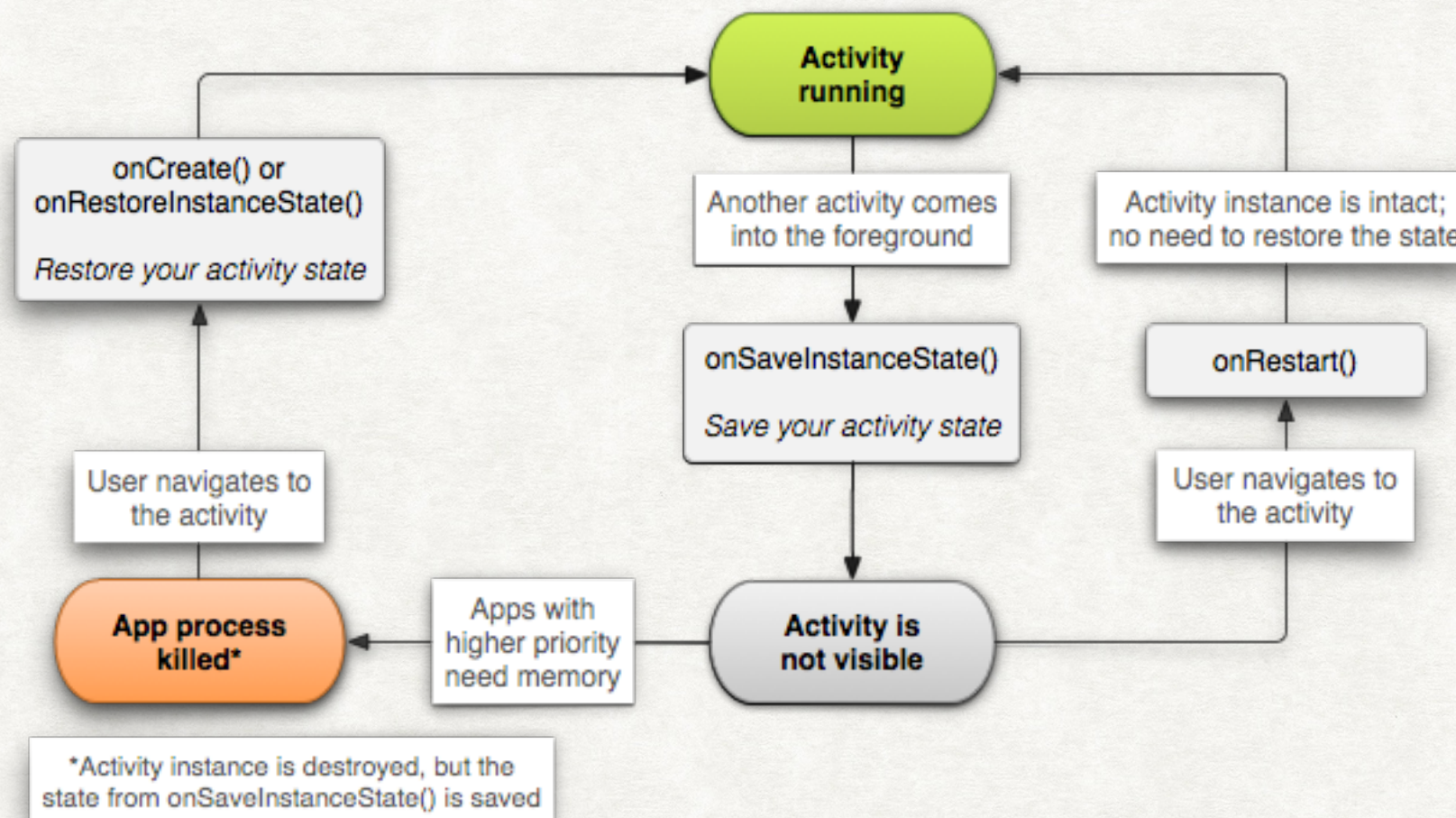
running - l'activité en avant plan

paused - l'activité en arrière plan
mais toujours partiellement visible

stopped - activité en arrière plan,
invisible

et `onSaveInstanceState(Bundle)` ?

onSaveInstanceState(Bundle bundle)



Sauvegarde de données persistantes

Pour sauvegarder les données permanentes dans une base de données, dans un fichier ou dans les préférences écrire le code qui effectue cette tâche dans la méthode :

`onPause()`

Si l'activité crée un thread pour charger les données depuis internet alors on arrête ce thread dans

`onDestroy()`

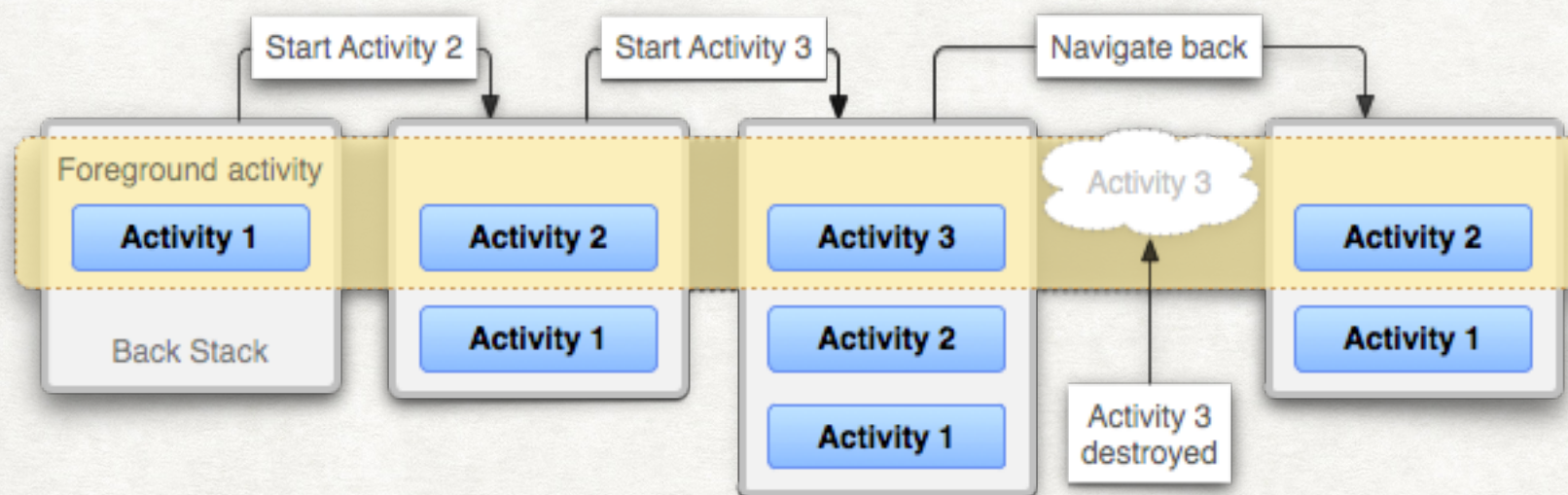
Coordination entre les activités

L'ordre d'exécution de méthodes quand l'activité A lance l'activité B :

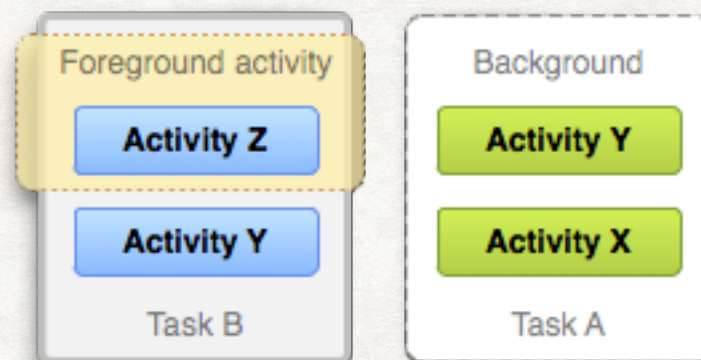
- 1.A exécute `onPause()`
- 2.B exécute `onCreate()`, `onStart()`,
`onResume()`
- 3.A exécute `onStop()`

Donc si A écrit dans une BD ou dans un fichier et B utilise les données écrites par A alors A doit tout écrire dans `onPause()`

Backstacks et les tasks



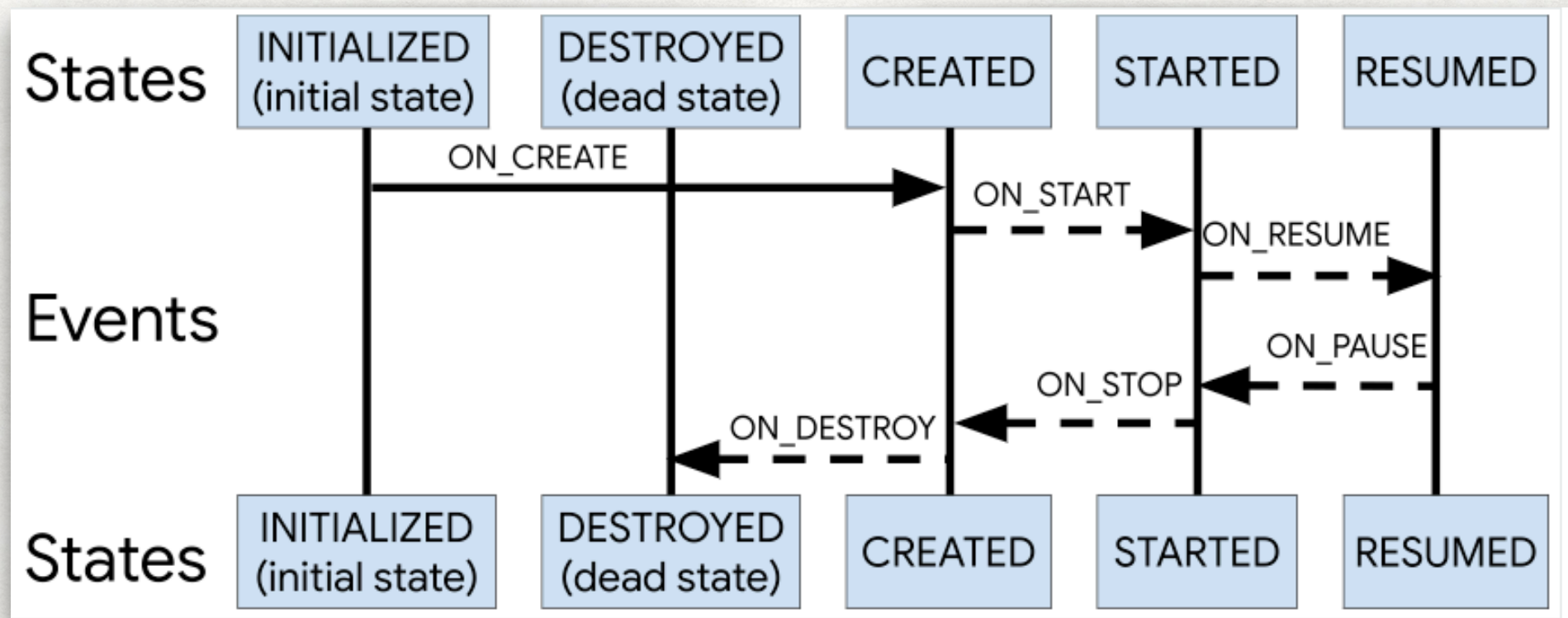
backstack d'un task



Deux tâches, chaque tâche avec son backstack

Cycle de vie et la classe Lifecycle

La classe **Lifecycle** maintient les informations sur le cycle de vie d'un composant comme Activity ou Fragment.



Etats et événements qui constituent le cycle de vie de l'activité

Si un composant implémente l'interface **LifecycleOwner** cela signifie que ce composant possède le cycle de vie. **Activity**, **Fragment** implémentent cette **LifecycleOwner**

Ajouter les dépendances dans build.gradle

Il faut ajouter dans build.gradle(Module) les dépendances indiquées sur la page suivante (pas toutes, ViewModel et LiveData suffisent dans la plupart de cas) :

https://developer.android.com/jetpack/androidx/releases/lifecycle#declaring_dependencies

Pas la peine d'ajouter toutes les dépendances, pour une application simple il suffit juste :

```
dependencies {  
    def lifecycle_version = "2.6.0-alpha02"  
    // ViewModel  
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version"  
    // LiveData  
    implementation "androidx.lifecycle:lifecycle-livedata-ktx:$lifecycle_version"  
}
```

J'ai utilisé gradle version 7.5.1 et Android gradle plugin version 7.3.0

qu'on spécifie dans File -> Project Structure -> Project

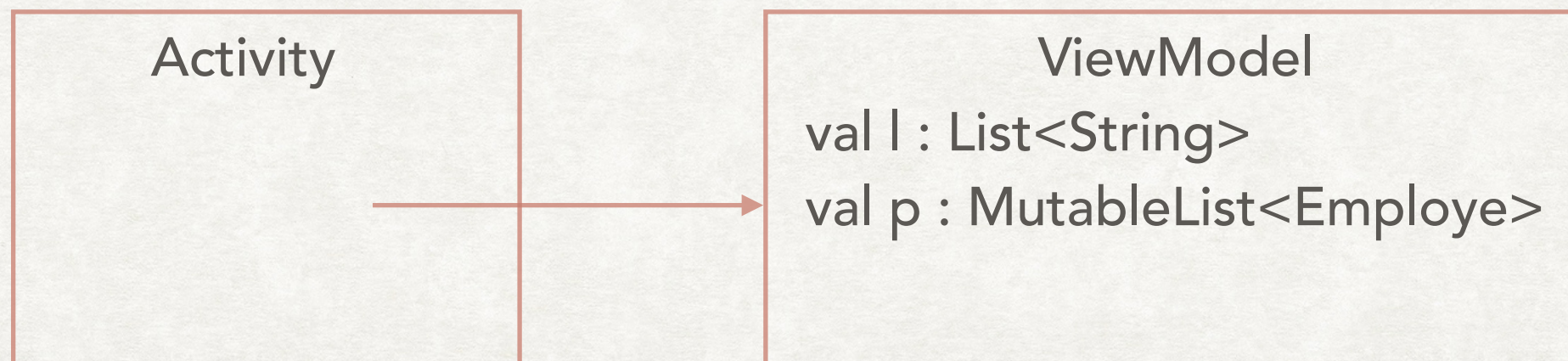
(mais peut-être cela marche aussi avec d'autres versions de gradle et Android plugin gradle).

ViewModel et LiveData

Problème de préservation de données de l'activité pendant le changement de configuration.

Remède : **ViewModel**

ViewModel est une classe permet de stocker les données d'une activité jusqu'à la destruction définitive de l'activité (jusqu'à l'exécution de `onDestroy()`). Après l'exécution de `onDestroy()` on ne revient plus jamais vers la même instance de cette activité. ViewModel n'est pas détruit pendant le changement de configuration. Donc il suffit de stocker toutes les données dans le ViewModel de l'activité et non pas dans l'activité elle même.



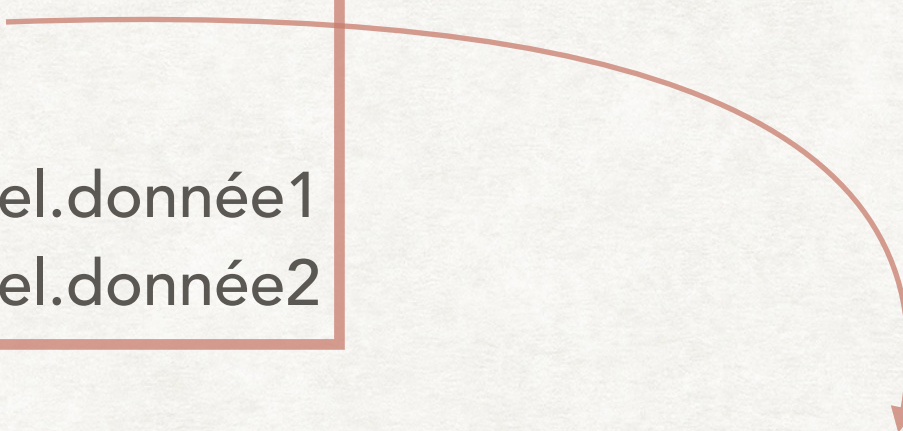
partage de rôle entre ViewModel et Activity

- Ne jamais stocker les données dans l'activité
- Les données et la logique du programme dans le ViewModel
- L'activité gère uniquement l'interface graphique et, éventuellement, modifie les données de ViewModel en fonction de l'action de l'utilisateur
- Dans le sens inverse, si les données changent alors l'activité aura peut-être à modifier l'interface graphique. L'activité installe les **Observers** sur les données de ViewModel. Les Observers sont déclenchés quand les données changent. Pour pouvoir observer les données, ces données doivent être de type **LiveData** ou **MutableLiveData**

partage de rôle entre ViewModel et Activity

Activity

```
val viewModel : MyViewModel  
  
installer Observer sur viewModel.donnée1  
installer Observer sur viewModel.donnée2
```



MyViewModdel : ViewModel()

```
val données1 : MutableLiveData<String>  
val données2 : LiveData<List<Int>>
```


Activity : comment récupérer la référence vers le ViewModel ?

première possibilité

L'activité doit trouver une référence vers mon implémentation de ViewModel. Dans mon application j'ai défini la classe MyViewModel dérivée de ViewModel qui sert de ViewModel de l'activité.

```
class MainActivity : AppCompatActivity() {  
  
    //récupérer la référence vers le ViewModel  
    private val model : MyViewModel by ViewModel()  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
    }  
}
```

Après le démarrage de l'activité Android crée le nouveau ViewModel implémenté par ma classe MyViewModel (voir les transparents suivants).

Après le changement de configuration il n'y a pas de création de nouveau ViewModel, l'attribut model sera réinitialisé avec une référence vers MyViewModel créé au lancement de l'activité.

Pour utiliser cette méthode il faut ajouter dans **build.gradle** dans la section **dependencies** la ligne suivante
implementation "androidx.activity:activity-ktx:1.6.0"

Activity : comment récupérer la référence vers le ViewModel ?

deuxième possibilité

Avec cette méthode pas besoin d'ajouter dans build.gradle la ligne

mentionnée sur le transparent précédent :

```
class MainActivity : AppCompatActivity() {
```

```
    //obtenir la référence vers le ViewModel
```

```
    private val viewModel by
```

```
        lazy{ ViewModelProvider(this).get(MyViewModel::class.java) }
```

```
    override fun onCreate(savedInstanceState: Bundle?) {
```

```
        super.onCreate(savedInstanceState)
```


Activity

Activity installe un observer sur les données stockées dans le ViewModel.

`androidx.lifecycle.Observer<T>`

est une interface avec une seule méthode :

`onChanged(T t) : Unit`

Donc pour implémenter Observer on utilisera une lambda expression à un seul paramètre.

Comment installer Observer sur **LiveData** ou **MutableLiveData** ?

Ces deux classe possèdent la méthode :

```
LiveData.observe( @NonNull owner: LifecycleOwner,  
                  @NonNull observer: Observer<in T>) : Unit
```

Le premier paramètre de observe() dans l'activité est l'activité elle même (elle est LifecycleOwner), donc souvent c'est **this**. Le deuxième c'est la lambda expression qui implémente **Observer**.

Activity

Activity installe un observer sur les données stockées dans le ViewModel. Dans l'exemple une seule donnée est un objet compteur de type LiveData<Int> qui stocke un Int.

L'observer est activé à chaque modification de contenu de LiveData (et aussi à la création de l'activité).

```
//installation  
model.compteur.observe(this) { binding.compteur.text = "$it" }
```

compteur est un **MutableLiveData<Int>** stocké dans le modèle. Quand la valeur de compteur change l'observer met à jour la valeur affichée dans le TextView dont id est "compteur".

L'observer est ici implémenté par une lambda expression. Le paramètre par défaut de lambda expression est **it** et c'est l'objet (Int) stocké dans **MutableLiveData**.

Activity

Activité installe aussi un listener (le même) sur les deux boutons. L'activation d'un bouton par utilisateur change la valeur du compteur de MyViewModel. (Au lieu de listener la méthode nommée par l'attribut onClick de boutons).

```
fun click(view: View) {  
    val i = when (view.id) {  
        R.id.ajouter -> 1  
        R.id.soustraire -> -1  
        else -> 0  
    }  
    //récupérer la valeur du compteur  
    val v = model.compteur.value ?: 0  
    // ajouter i à la valeur de compteur et remettre and  
MutableLiveData  
    // la nouvelle valeur  
    model.compteur.setValue(v + i)  
}
```


Activity

```
class MainActivity : AppCompatActivity() {  
  
    private lateinit var model: MyViewModel  
    private val binding : ActivityMainBinding by  
        lazy{ActivityMainBinding.inflate( inflater )}  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
        super.onCreate(savedInstanceState)  
  
        setContentView(binding.root)  
  
        // trouver la référence vers le ViewModel  
        model = ViewModelProvider(this).get(MyViewModel::class.java)  
  
        //installer observer sur la donnée de ViewModel  
        model.getCompteur().observe(this){ binding.compteur.text = "$it" }  
    }  
  
    fun click(view: View) {  
        val i = when (view.id) {  
            R.id.ajouter -> 1  
            R.id.soustraire -> -1  
            else -> 0  
        }  
        //récupérer la valeur du compteur  
        val v = model.getCompteur().value ?: 0  
        model.getCompteur().setValue(v + i)  
    }  
}
```

↑
it : le paramètre de lambda = la valeur (Int)
contenue
dans l'objet MutableLiveData<Int>

model.getCompteur() retourne
un objet MutableLiveData<Int>.
La propriété value est le Int englobé
par ce MutableLiveData

LiveData

Un `LifecycleOwner` (activité) enregistré sur un objet `LiveData` ou `MutableLiveData` est informé de changement de valeur de l'objet stocké dans `LiveData` via une fonction callback. La fonction callback est activée uniquement quand `LifecycleOwner` qui est enregistré en tant que `Observer` est en état de recevoir la nouvelle valeur, c'est-à-dire quand l'activité est visible. `Observer` qui n'est pas actif n'obtient pas de données.

Comment travailler avec `LiveData` ?

- Créer une instance de `LiveData` (sans doute dans un `ViewModel`)
- Créer un `Observer` qui implémente la méthode `onChanged()`. D'habitude l'observer est créé dans une activité ou dans un `Fragment`.
- Attacher l'observer to `LiveData` avec la méthode `observe()`.

Remarque. Il est possible d'attacher un observer qui n'est pas un `LifecycleOwner` en utilisant la fonction

`observeForever(Observer)`.

Un tel `Observer` peut être enlevé par l'appel à `removeObserver(Observer)`.

LiveData

//MyViewModel est une extension de ViewModel

```
class MyViewModel : ViewModel() {  
  
    private val compteur: MutableLiveData<Int>( 0 )  
    fun getCompteur() = compteur  
  
}
```


COMMENT RÉCUPÉRER LES RESSOURCES DANS VIEWMODEL

AndroidViewModel

Parfois pour initialiser les attributs de ViewModel nous avons besoin de lire les ressources.

Nous sommes devant un problème :

- pour lire les ressources il faut le contexte, et le contexte utilisé c'était Activity
- ViewModel ne doit en aucun cas contenir une référence vers Activity. Cela peut provoquer les fuites de mémoire et même des erreurs à l'exécution de l'application.

Ces deux conditions semblent contradictoires. Mais il existe une autre possibilité d'obtenir le contexte, application elle-même est aussi un contexte. Et l'application est bien vivante tant que au moins un composant de l'application est vivant.

AndroidViewModel est comme ViewModel mais avec un constructeur qui a comme paramètre application.

AndroidViewModel

```
class MyViewModel(application: Application) :  
    AndroidViewModel(application) {  
  
    private val allColors: MutableLiveData<MutableList<String>>  
  
    init {  
        // initialiser allColors depuis les ressources  
        val c = application.resources  
            .getStringArray(R.array.colors)  
            .toMutableList()  
  
        allColors = MutableLiveData<MutableList<String>>(c)  
        //notez au passage qu'on peut initialiser  
        //l'attribut val dans le bloc init  
    }  
    ...  
}
```

Dans ce fragment de code MyViewModel est dérivé de AndroidViewModel. Le paramètre est l'application. L'application possède une référence vers les ressources : voir le code en gras.

Il n'y a aucune différence par rapport à l'utilisation de ViewModel et AndroidViewModel du côté de Activity.