ANDROID

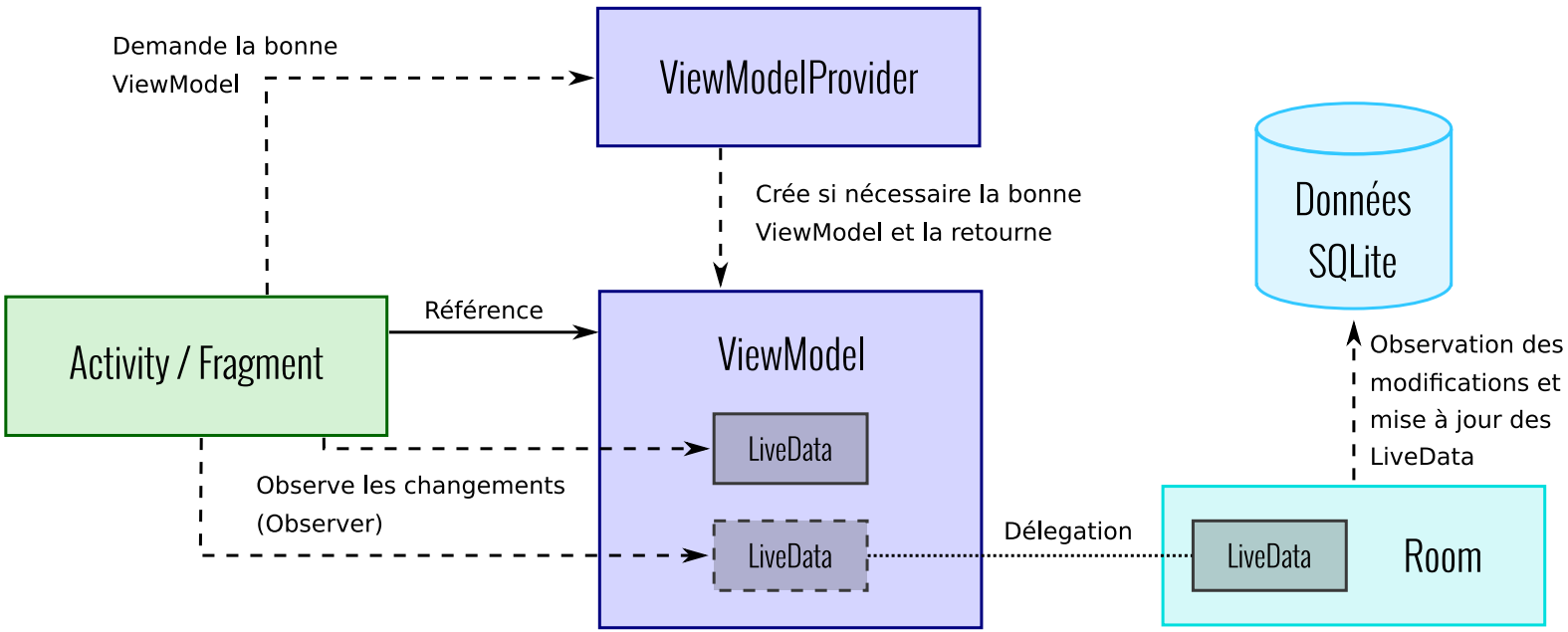
Laurent Provot <laurent.provot@uca.fr>

Janvier 2023

Android Jetpack : ViewModel et LiveData

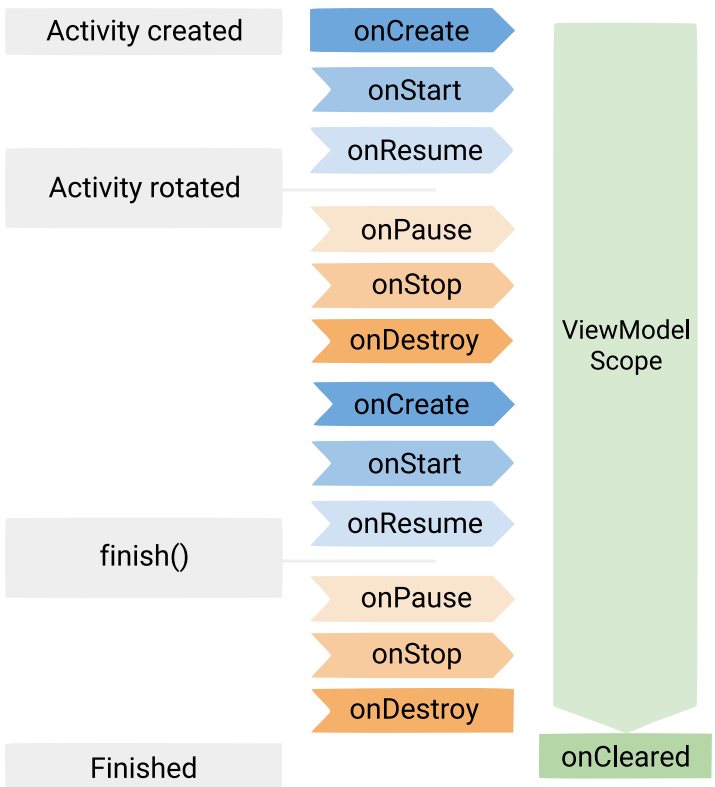
Les composants de l'architecture

- Gestion des données : `Room` + `LiveData` + `ViewModel`
- Information chargée en asynchrone (`Room` + `LiveData`)
- Survie en cas de changement de configuration (`ViewModel`)
- Mécanisme d'observation de la donnée (`LiveData`)



ViewModel

- Permet de stocker les données utiles à l'UI
- Survie à des changements de configuration, mais pas à des terminaisons «normales»



<https://developer.android.com/topic/libraries/architecture/viewmodel>

`ViewModel`

- Classe qui dérive de `ViewModel` (ou `AndroidViewModel` si besoin d'un contexte)
- Ne doit surtout PAS référencer des éléments de l'UI (`Context`, `View`, `Fragment`, …)

```
class MyViewModel : ViewModel() {
    val myData : LiveData<Data> = …

    fun loadData() { … }
    fun saveData(data: Data) { … }
    fun isSomething() { … }
}
```

- Expose les données nécessaires à la vue (`LiveData`, …)
- Expose les méthodes d'accès aux données

ViewModel

- Récupération au sein d'un fragment ou d'une activité

```
class MyFragment : Fragment() {
    private lateinit var myViewModel: MyViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        …
        myViewModel = ViewModelProvider(this)
                        .get(MyViewModel::class.java)

    }
}
```

- Et si la `ViewModel` a besoin de paramètres ?

```
class DogViewModel(val dogId: Long) : ViewModel() { … }
```

ViewModelFactory

- Le `ViewModelProvider` utilise une factory pour instancier la `ViewModel`
- Création d'une factory personnalisée, puis passage au `ViewModelProvider`

```
class DogVMFactory(private val dogId: Long) : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T {
        return DogViewModel(dogId) as T
    }
}

class DogFragment : Fragment() {
    private lateinit var dogVM: DogViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val dogId = arguments?.getLong(EXTRA_DOG_ID) ?: NEW_DOG_ID
        dogVM = ViewModelProvider(this, DogVMFactory(dogId))
                            .get(DogViewModel::class.java)
    }
}
```

ViewModelFactory

- Pas mal de boilerplate code (surtout si plein de VM $\neq$)
- Heureusement, vive Kotlin

```
@Suppress("UNCHECKED_CAST")
inline fun <VM : ViewModel> viewModelFactory(crossinline f: () -> VM) =
object : ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass: Class<T>): T = f() as T
}
```

- Et au sein du fragment (ou activité)

```
dogVM = ViewModelProvider(this, viewModelFactory { DogViewModel(dogId) })
                          .get(DogViewModel::class.java)
```

ViewModelFactory

- Pour ceux qui préfèrent utiliser les extensions KTX
- Dans le build.gradle

```
implementation "androidx.fragment:fragment-ktx:x.y.z"
```

- Fragment KTX offre des *property delegate*

1. Pour les cas simples

```
val viewModel by viewModels<MyViewModel>()
```

2. Pour les cas avec paramètres

```
val viewModel by viewModels<DogViewModel> { DogVMFactory(dogId) }
```

`LiveData`

- Conteneur de donnée : `LiveData<T>`
- Permet d'ajouter des observateurs sur `T`
- Les observateurs sont notifiés dès que `T` change

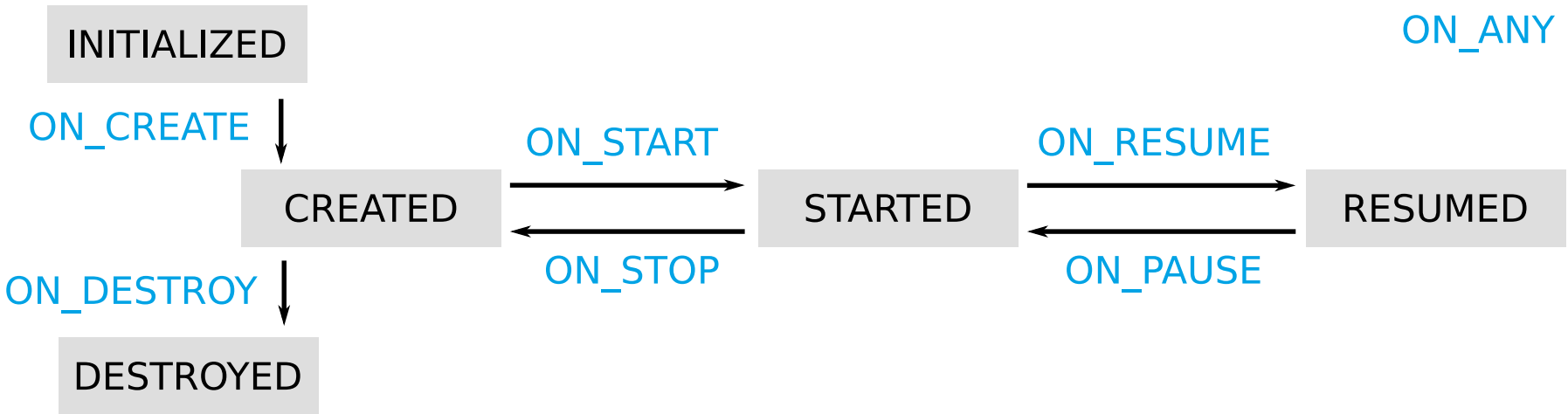
```
class DogFragment : Fragment() {
    private lateinit var dogVM: DogViewModel

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        val dogId = arguments?.getLong(EXTRA_DOG_ID) ?: NEW_DOG_ID
        dogVM = ViewModelProvider(this, DogVMFactory(dogId))
                            .get(DogViewModel::class.java)
        dogVM.dog.observe(this, Observer { updateFragmentContent(it) })
    }
}
```

- Que se passe-t-il si l'observateur est appelé quand la vue n'est pas affichée ?
- Faut-il supprimer les observateurs pour éviter les problèmes ?

`Lifecycle`

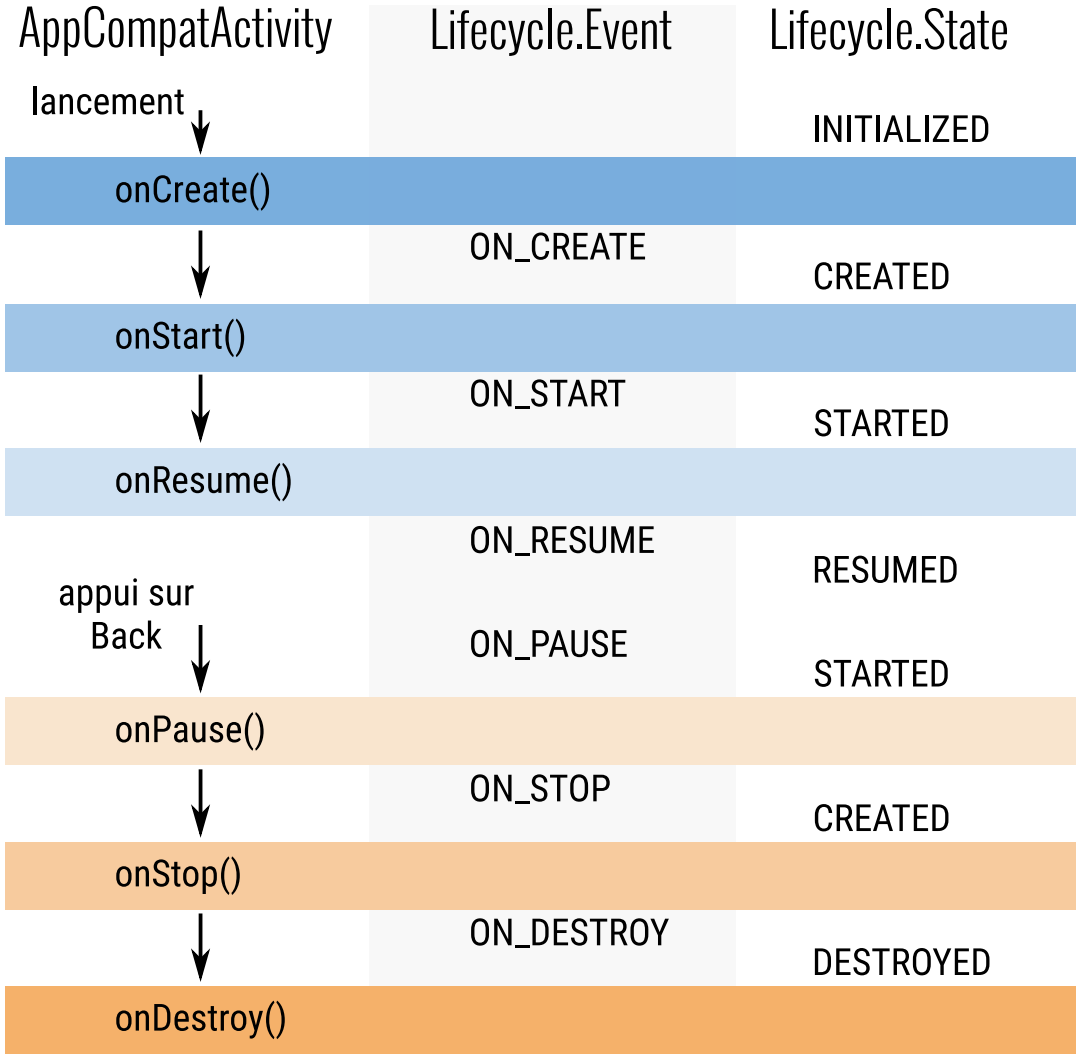
- Les `LiveData` et `ViewModel` sont des composants qui sont «Lifecycle aware»
- Typage de la notion de cycle de vie avec Jetpack
- Introduction de `Lifecycle`, `LifecycleOwner` et `LifecycleObserver`
- Machine à état `Lifecycle.State` + `Lifecycle.Event`



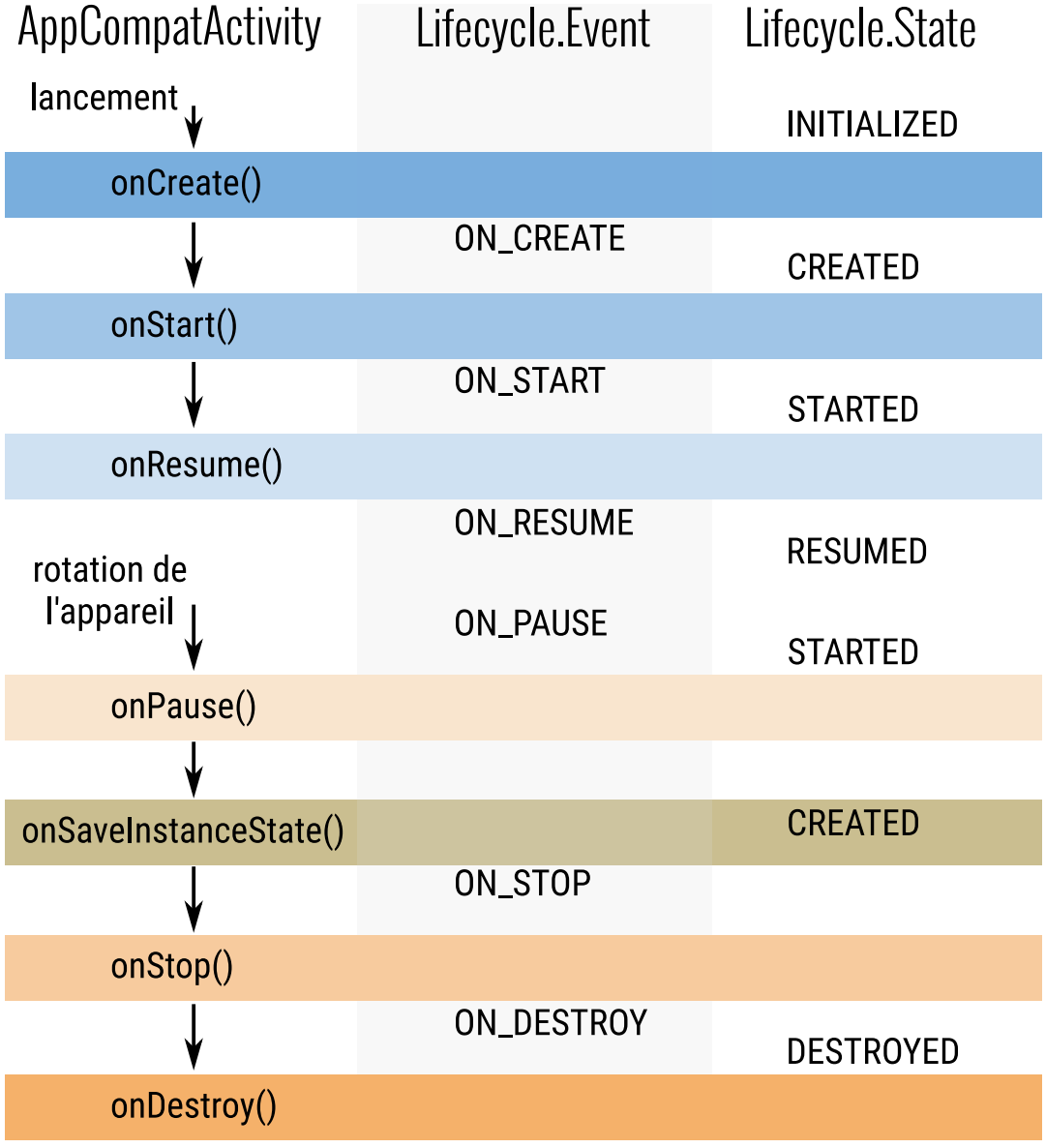
`LifecycleOwner`

- Les entités possédant un `Lifecycle` implémentent `LifecycleOwner`
- N'expose que la méthode `getLifecycle()` qui retourne le `Lifecycle`
- Synchronise les méthodes du cycle de vie du composant avec les états du `Lifecycle`
- Cas légèrement différent suivant :
  1. terminaison normale (`finish()`)
  2. changement de configuration (rotation de l'appareil)

`LifecycleOwner` (terminaison)



`LifecycleOwner` (changement config)



`LifecycleObserver`

- Possibilité d'observer et de réagir aux `Lifecycle.Event`

```
class MyObserver : LifecycleObserver {
    @OnLifecycleEvent(ON_RESUME)
    fun startMonitoring(src: LifecycleOwner) { … }

    @OnLifecycleEvent(ON_PAUSE)
    fun stopMonitoring() { … }

    @OnLifecycleEvent(ON_ANY)
    fun onAny(src: LifecycleOwner, evt: Event) { … }
}
```

- Dans le `LifecycleOwner`

```
lifecycle.addObserver(MyObserver())
```

`LiveData` (suite)

- Les `LiveData` utilisent des `LifecycleObserver` pour être «Lifecycle aware»
- Elles ne notifient leurs observateurs que pour un lifecycle actif
- Lifecycle actif = dans l'état `STARTED` ou `RESUMED`
- Les observateurs sont automatiquement supprimés au passage à l'état `DESTROYED` (évite les fuites mémoires)
- Différentes implémentations de `LiveData` existantes :
  - `MutableLiveData` : on peut modifier la valeur
  - `MediatorLiveData` : réaction en fonction de plusieurs sources
  - `Transformations.map()` et `switchMap()` : conversion classiques de `LiveData`

`MutableLiveData`

- `LiveData` : classe abstraite, non instanciable directement
- Getter pour la valeur + méthodes de gestion des observateurs
- `MutableLiveData` expose en plus 2 setter :
  1. `setValue(T)` pour modifier la valeur depuis le Thread UI
  2. `postValue(T)` pour modifier la valeur n'importe où
- `MutableLiveData` est une classe concrète, donc instanciable
- Si `MutableLiveData` dans VM, n'exposer qu'une version `LiveData` si possible

```
private val _dog: MutableLiveData<Dog>()
val dog: LiveData<Dog>
  get() = _dog
```

MediatorLiveData

- Permet d'enregistrer plusieurs `LiveData` sources
- Engendre une action dès que l'une d'entre elles change

```
val networkErrors: LiveData<String> = …
val monitoringLogs: LiveData<LogLine> = …

val messages = MediatorLiveData<String>()

messages.addSource(networkErrors) { messages.postValue("Error: $it") }
messages.addSource(monitoringLogs,
                   { line -> messages.postValue("Log: ${line.value}") })
```

Transformations

- Une donnée de type `T` gérée à travers une `LiveData<T>`
- L'UI a besoin d'afficher une donnée de type `U` synchronisée aux changements de la donnée `T` précédente
- Plutôt que de créer et faire une nouvelle requête avec la DAO utiliser `Transformations.map()`
  - Prend une `LiveData<T>` source et une fonction `T -> U`
  - Retourne une `LiveData<U>` en appliquant la fonction sur la valeur de la source
- Attention, la transformation se fait sur le Thread UI

Exemple : on veut afficher le nom d'un chien et son poids entre parenthèses

```
val dog: LiveData<Dog> = dogRepo.findById(dogId)
val shortDesc: LiveData<String> = Transformations.map(dog) {
    "${it.name} (${it.weight})"
}
```

Transformations

- PAS de transformations lourdes avec `map()`
- Possibilité d'utiliser `Transformations.switchMap()`
- Semblable à `map()` mais en commutant des `LiveData`
- `Transformations.switchMap()`
  - Prend une `LiveData<T>` source et une fonction `T -> LiveData<U>`
  - Retourne une `LiveData<U>` qui est synchronisée sur celle retournée par la fonction précédente quand la source est modifiée

Exemple : on veut une `LiveData<Dog>` qui se met à jour quand une `dogId: LiveData<Long>` change

```
val dogId: LiveData<Long> = MutableLiveData()
val dog: LiveData<Dog> = Transformations.switchMap(dogId) {
    dogRepo.findById(it)
}
```