ANDROID

Laurent Provot <laurent.provot@uca.fr>

Janvier 2023

Android Jetpack : le Data Binding

Data Binding

- Synchroniser la donnée affichée avec la donnée du modèle
- Éviter le boilerplate code au sein des `Fragment` / `Activity`
- Déclaration au sein des layouts XML plutôt qu'en code
- Fonctionnalités :
  - Binding one-way et two-way
  - Expressions de binding
  - Fonctionne de pair avec les `Observable` (et `LiveData` et plus récemment `StateFlow`)
  - Binding adapters pour la personnalisation de la logique
- Malheureusement pas beaucoup d'aide du compilateur pour les erreurs

Mise en place au sein du projet

- Disponible seulement pour API >=14 (Andoid 4.0 — Ice Cream Sandwich)
- Activation dans le `build.gradle`

```
android {
  …
  buildFeature {
    dataBinding true
  }
  …
}
```

Au niveau de la ressource XML

- Entourer le layout « classique » dans un balise `<layout>`
- Déclarer les variables utilisées au sein du layout dans une balise `<data>`

```
<?xml version="1.0" encoding="utf-8"?>
<layout xmlns:android="http://schemas.android.com/apk/res/android" …>

    <data>
        <variable name="dogVM" type="ouafff.ui.DogViewModel"/>
    </data>

    <androidx.constraintlayout.widget.ConstraintLayout>
    …
    </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

Au sein du layout lui même

- Ajouter les liaisons entre les vues et les variables :
  - `"@{…}"` pour un binding one-way
  - `"@={…}"` pour un binding two-way
- Possibilité de mettre une valeur par défaut pour le designer
  - `android:text="@{variable, default=@string/key}"`

```
<EditText
    android:id="@+id/edit_dog_name"
    android:text="@={dogVM.dog.name}"
    android:inputType="textCapWords" …/>
<TextView
    android:id="@+id/text_dog_owner"
    android:text="@{dogVM.dog.owner,
                  default=@string/def_owner}" … />
```

Expressions de binding

- Possibilité de créer des expressions si binding pas direct

```
<TextView
    android:id="@+id/text_fullname"
    android:text="@{user.firstName + ' ' + user.lastName}"/>
```

- Opérateurs mathematiques (`+` , `*` , ...), opérateurs logiques (`||` , `>>` , ...), comparaisons, littéraux, cast, appels de méthodes, `[]` , `()` , operateur ternaire (`?:`)
- Pas possible : `this` , `super` , `new`
- Valeur en cas de `null` : `android:text="@{user.pseudo ?? 'John Doe'}"`
- Attention : garder les expressions de binding simples, sinon intérêt limité

Binding pour les Event Handler

- Possibilité de spécifier un event handler simple directement en XML
- Attention : à reserver pour les appels triviaux
- Pour les réferences de méthodes, les signatures doivent correspondrent

```
class DogViewModel … {
    fun nextDog() { … }
    fun update(view : View) { … }
}
```

```
<data><variable name="dogVM" type="DogViewModel"/></data>

<com.google.android.material.button.MaterialButton
            android:id="@+id/button_next_dog"
            android:onClick="@{() -> dogVM.nextDog()}">
<com.google.android.material.button.MaterialButton
            android:id="@+id/button_update"
            android:onClick="@{dogVM::update}">
```

Classe de Binding

- Pour chaque fichier layout (`item_layout_view.xml` par exemple) une classe pour gérer le data binding est générée automatiquement
- Nom par défaut : nom du fichier layout en Pascal Case sans les underscores suffixé par Binding (par exemple `ItemLayoutViewBinding`)
- Pour un module dont le package est `app.module.pkg` , la classe est générée dans `app.module.pkg.databinding`
- Pour modifier ça utiliser l'attribut `class` de la balise `<data>`

```
<data class="app.pkg.other.CustomName">
    ...
</data>
```

Classe de Binding

- Les id des ressources utiles au binding sont gérés au sein de la classe `BR` (équivalent de la classe `R` mais uniquement pour le data binding)
- Chaque vue qui possède un id (`android:id="@+id/my_view"` par exemple) est disponible à travers une propriété au sein de la classe de binding générée
- Nom de la propriété : l'id en Camel Case sans les underscores (`myView` par exemple)
- Utile car évite de faire des `findViewById()`
- On préférera utiliser les propriétés de la classe de binding générée

Récupération au sein de l'UI

- Mise en place effective du binding au sein de l'UI

1. Inflate du layout (ou remplacement du `setContentView()`)
2. Initialisation des variables du binding
3. Attention : si `LiveData` ne pas oublier de specifier le `LifecycleOwner`

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    val vm = ViewModelProviders.of(this).get(DogViewModel::class.java)
    val binding = DataBindingUtil.setContentView(this,
                                                 R.layout.fragment_dog)
    binding.dogVM = vm
    binding.lifecycleOwner = this
}
```

Récupération au sein de l'UI

- Pour un fragment : pas de `setContentView()`

```
override fun onCreateView(inflater: LayoutInflater, container: ViewGroup?,
                          savedInstanceState: Bundle?): View? {
    val binding = FragmentDogBinding.inflate(inflater)
    binding.dogVM = viewModel
    binding.lifecycleOwner = viewLifecycleOwner
    return binding.root
}
```

- Possiblilité de séparer la création de la vue et la mise en place du binding

```
val view = LayoutInflater.from(this).inflate(layoutId, parent, attachToParent)
val binding: ViewDataBinding? = DataBindingUtil.bind(view)
```

Personnalisation du data binding

- Soit le cas de figure général suivant :

```
<pkg.MyView app:myAttribut="@{value}" />
```

- Quand `value` change, la vue `MyView` doit se mettre à jour
- Comportement par défaut : une méthode `setMyAttribut(arg)` est appelée sur l'instance de `MyView` où `arg` doit être du même type que `value`

- Que faire quand les conditions d'application de ce comportement ne sont pas vérifiées ?

Personnalisation du data binding

A. Que faire quand `value` et le paramètre de `setMyAttribut(…)` ne sont pas du même type ?
B. Que faire quand `setMyAttribut(…)` n'existe pas dans `MyView` ?
C. Que faire quand `setMyAttribut(…)` n'implémente pas le comportement désiré ?

- Réponse : on utilise des *binding adapters*

Cas A : conversion de type

- Les types des propriétés et données bindées doivent correspondre
- Sinon utiliser des méthodes de conversion : les *converters*
- Méthodes statiques (`@JvmStatic` ou top-level en kotlin) annotées

Cas A : conversion de type

1. Converters implicites
   - Méthode `xToY(x: X): Y` annotées `@BindingConversion`

```
@BindingConversion
fun booleanToVisibility(isVisible: Boolean) =
    if (isVisible) View.VISIBLE else View.GONE
```

   - Automatiquement choisie dès qu'une conversion `Boolean` en `Int` est nécessaire
   - Attention donc, comportement global, très risqué
   - Par exemple sera utilisée dans une `RatingBar` pour le binding `android:rating=@{true}`, ce qui n'a aucun sens

Cas A : conversion de type

2. Converters explicites
   - Méthode `xToY(x: X): Y` : implémente la conversion d'un `X` en `Y`
   - Pour du DB two-way implémenter aussi la méthode qui effectue la conversion dans l'autre sens : `yToX(y: Y): X`
   - Et annoter `xToY()` avec `@InverseMethod("yToX")`
   - Expliciter l'utilisation lors du binding :

```
<data> …
    <import type="app.pkg.utils.Converters" />
</data>

<EditText
    android:id="@+id/edit_dog_weight"
    android:text="@={Converters.floatToString(dogVM.dog.weight)}"
    … />
```

Cas A : conversion de type

- Le code associé sera alors le suivant :

```
package app.pkg.utils

object Converters {
    @JvmStatic
    @InverseMethod("stringToFloat")
    fun floatToString(value: Float) = value.toString()

    @JvmStatic
    fun stringToFloat(value: String) = if (value.isBlank())
                                           0f else value.toFloat()
}
```

Cas B : spécification d'une autre méthode pour le binding

- Si on veut utiliser la méthode `setCustomAttrMethod(…)` déjà existante car `setMyAttibut(…)` n'existe pas
- Annoter une classe avec `@BindingMethods` et spécifier l'association avec `BindingMethod`

```
@BindingMethods(value = [
    BindingMethod(type = app.pkg.MyView::class,
                  attribute = "app:myAttribut",
                  method = "setCustomAttrMethod")
])
class MyAdapter { … }
```

Cas C : implémentation d'une nouvelle méthode pour le binding

- S'il n'existe pas de méthode adéquate à appeler
- Implémenter une nouvelle méthode en l'annotant `@BindingAdapter` et spécifier à quel attribut elle s'applique

```
@BindingAdapter("app:myAttribut")
fun setCustomMyAttr(view: MyView, oldValue: XXX, newValue: XXX) {
    // traiement à faire pour mettre à jour view
    // en fonction de oldValue et newValue
}
```

- Il est possible de ne pas mettre `oldValue: XXX` si elle n'est pas utilisée