DAA - Développement d'applications Android

Interface Graphique

28 October 2025

Table des matières

1 ListView et RecyclerView

1.1 ListView

La ListView est un composant permettant d'afficher verticalement une liste (cohérente) d'éléments. La ListView ne connaît ni le type ni le nombre d'éléments qu'elle doit afficher. Elle s'appuie sur un adaptateur (Adapter) pour obtenir les données à afficher.

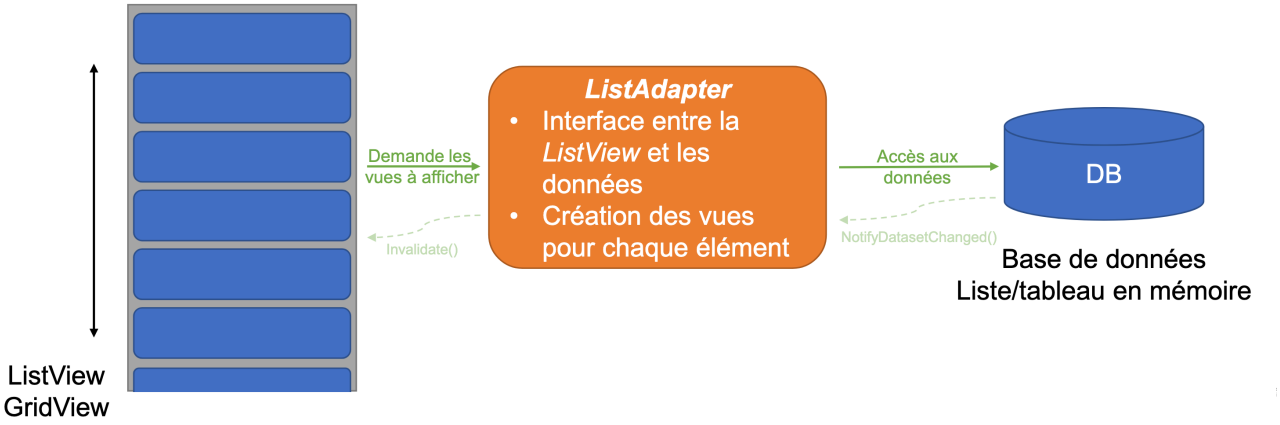

Fig. 1. – Capture des slides du cours – Fonctionnement d'une ListView

1.1.1 Adapteur (Adapter)

L'adaptateur est une interface entre la ListView et les données à afficher. Il permet de fournir les vues pour chaque élément de la liste en fonction des données sous-jacentes.

1.1.1.1 API minimale

Pour utiliser une ListView, il est nécessaire de créer une classe qui étend l'adaptateur de base (BaseAdapter) et de surcharger les méthodes suivantes :

```
override fun getCount(): Int {
    // Retourne le nombre d'éléments dans la liste
}

override fun getItem(position: Int): Any {
    // Retourne l'élément à la position spécifiée
}

override fun getItemId(position: Int): Long {
    // Retourne l'ID de l'élément à la position spécifiée
}

override fun getView(position: Int, convertView: View?, parent: ViewGroup?): View {
    // Retourne la vue pour l'élément à la position spécifiée
}
```

1.1.2 Recyclage des vues

Pour optimiser les performances, la ListView réutilise les vues des éléments qui ne sont plus visibles à l'écran. Cela permet de réduire le nombre de créations de vues et d'améliorer la fluidité du défilement.

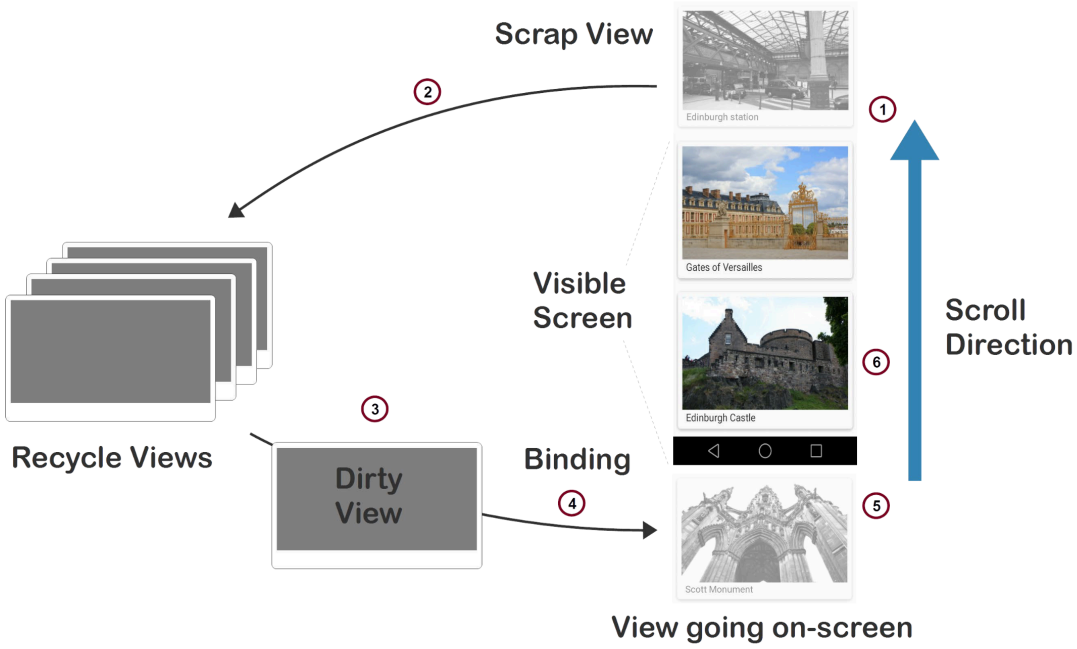

Fig. 2. – Capture des slides du cours – Recyclage des vues dans une ListView

- la méthode `notifyDataSetChanged()` doit être appelée sur l'adaptateur lorsque les données sous-jacentes changent, afin de mettre à jour la ListView.

1.1.3 Exemple d'une ListView

activity_main.xml:

```
<RelativeLayout xmlns:android="http://
schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/
tools"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">

  <ListView
    android:id="@+id/list"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />

</RelativeLayout>
```

list_item.xml

```
<RelativeLayout xmlns:android="http://
schemas.android.com/apk/res/android"
android:layout_width="match_parent"
android:layout_height="?
listPreferredItemHeight"
android:padding="2dp">

  <TextView
    android:id="@+id/list_item_title"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:layout_centerVertical="true" />
</RelativeLayout>
```

1.1.4 Activité liée à la ListView

```
class MainActivity : AppCompatActivity() {

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

    val list = findViewById<ListView>(R.id.list)
    val adapter = MyAdapter()
    list.adapter = adapter

    adapter.items = listOf("Chat", "Chien", "Vache", "Mouton", "Poule", "Cheval", "Dinde", "...")

    list.setOnItemClickListener { _, _, position, _ ->
      Toast.makeText(this, adapter.getItem(position), Toast.LENGTH_SHORT).show()
    }
```

```
    }
}
```

1.1.5 Optimisations

L'appel à la méthode `findViewById` est coûteux en termes de performances. Pour éviter des appels répétés lors du recyclage des vues, il est recommandé d'utiliser le pattern ViewHolder.

Celle-ci permet de stocker les références aux vues dans un objet dédié, évitant ainsi des appels redondants à `findViewById`.

1.1.6 Limitations

- Les ListView ne sont pas officiellement dépréciées, mais leur utilisation est déconseillée au profit des RecyclerView, qui offrent plus de flexibilité et de meilleures performances.
- Globalement les RecyclerView sont plus performantes et flexibles que les ListView, notamment pour les listes complexes ou de grande taille.

1.2 RecyclerView

La RecyclerView est un composant plus avancé et flexible que la ListView pour afficher des listes d'éléments. Elle offre une meilleure gestion des performances et une plus grande personnalisation.

1.2.1 Vues différentes

Contrairement à la ListView, la RecyclerView permet d'afficher différents types de vues dans la même liste, en utilisant des ViewHolders spécifiques pour chaque type d'élément.

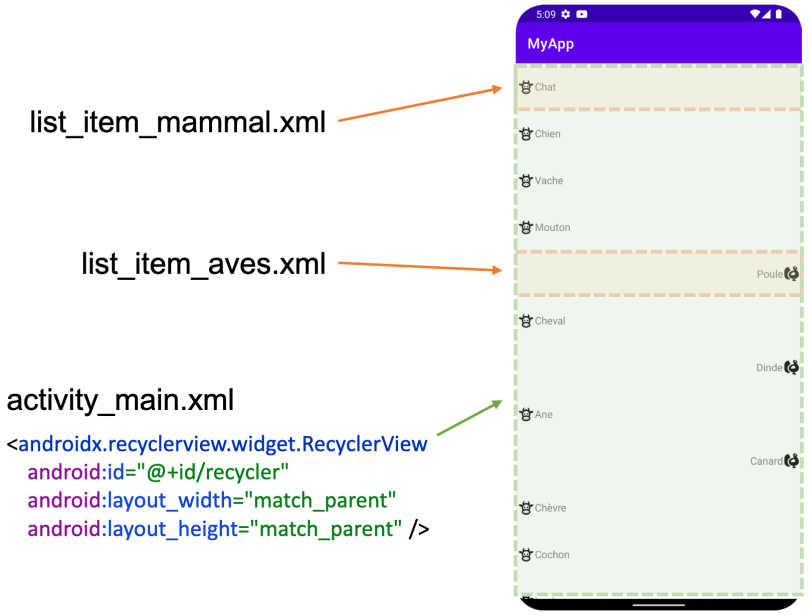

Fig. 3. – Capture des slides du cours – Différents types de vues dans une RecyclerView

1.2.2 Activité liée à la RecyclerView

```
class MainActivity : AppCompatActivity() {

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

    val recycler = findViewById<RecyclerView>(R.id.recycler)
    val adapter = MyRecyclerAdapter()
    recycler.adapter = adapter
    recycler.layoutManager = LinearLayoutManager(this)

    adapter.items = listOf( Mammal("Chat"), Mammal("Chien"), […], Mammal("Lapin"))
```

```
    }
}
```

Comparé une ListView, nous devons ajouter un paramètre `layoutManager` à la RecyclerView pour définir la disposition des éléments (par exemple, linéaire, grille, etc.).

1.2.3 Adapter pour RecyclerView

L'adaptateur pour une RecyclerView doit étendre `RecyclerView.Adapter` et utiliser des ViewHolders pour gérer les vues des éléments.

Une des grosse différence est que les RecyclerView permettent de rafraîchir uniquement une partie de la liste. Pour cela on implémente la méthode `DiffUtil` qui compare les anciennes et nouvelles données et met à jour uniquement les éléments modifiés.

```
class AnimalsDiffCallback(private val oldList: List<Animal>, private val newList: List<Animal>) :
DiffUtil.Callback() {

  override fun getOldListSize() = oldList.size

  override fun getNewListSize() = newList.size

  override fun areItemsTheSame(oldItemPosition: Int, newItemPosition: Int): Boolean {
    return oldList[oldItemPosition].id == newList[newItemPosition].id
  }
  override fun areContentsTheSame(oldItemPosition : Int, newItemPosition : Int): Boolean {
    val old = oldList[oldItemPosition]
    val new = newList[newItemPosition]
    return old::class == new::class && old.name == new.name
  }
}
```

Les différences possibles sont:

- `itemMoved` : un élément a été déplacé
- `itemRangeChanged` : le contenu d'un ou plusieurs éléments a changé
- `itemRangeInserted` : un ou plusieurs éléments ont été insérés
- `itemRangeRemoved` : un ou plusieurs éléments ont été supprimés