TP2 - Éléments Graphiques avancés : Adaptateurs & Listes

Télécharger PDF

[../tp2.pdf]

Objectifs du TP

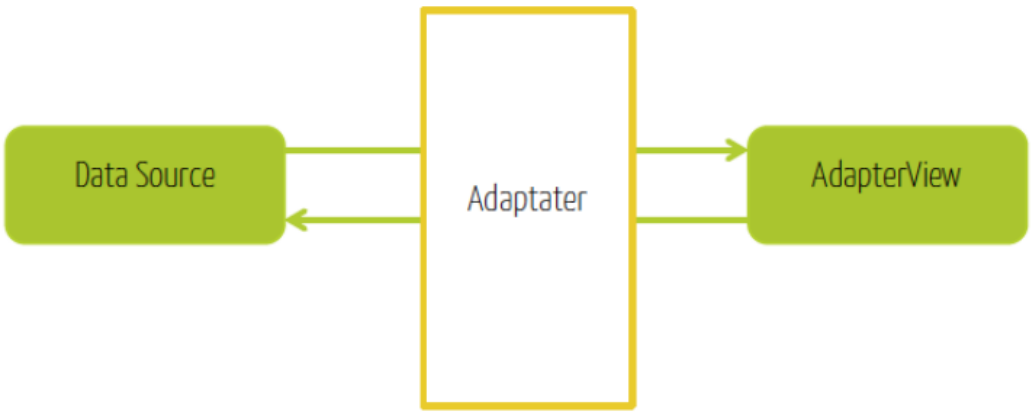
Expliquer la notion d'adaptateur via des exemples de listes, astuces d'utilisation, performance...

Outils et Versions

- Android Studio [https://developer.android.com/studio/index.html#tos-header] Version: 3.0.1
- Android : Version 5.0.1 (API 21)

I. Les Adaptateurs

Un objet adaptateur agit comme un pont entre une vue (appelée AdapterView) et les données sous-jacentes de cette vue. Une AdapterView est une vue complexe qui contient plusieurs vues, et qui nécessite un adaptateur pour construire ses vues filles.
Les AdapterViews les plus connues sont : **ListView**, **GridView** et **Spinner**.
Un adaptateur permet de transformer les données d'un DataSource (un tableau ou un
curseur) en widgets pour les insérer dans une AdapterView.



La méthode getView() de la classe Adapter permet de retourner, pour chaque élément de l'AdapterView, un objet View. Cette méthode est appelée à chaque fois qu'un élément de la liste va être affiché à l'écran.
Les deux types d'adaptateurs les plus communs sont : **ArrayAdapter** et **SimpleCursorAdapter**.

1. ArrayAdapter

Utiliser cet adaptateur quand votre source de données est un tableau. Par défaut, cet adaptateur crée une vue pour chaque élément du tableau en appelant la méthode toString de chaque élément, et en plaçant son contenu dans un objet TextView.
Un adaptateur de type ArrayAdapter est créé comme suit :

```
ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
      android.R.layout.simple_list_item_1,myStringArray);
```

Les arguments du constructeur sont :

- Le contexte de l'application (représente en général l'activité dans laquelle la AdapterView est créée)
- Un layout représentant le type d'un élément de la liste : Ce layout **android.R.layout.simple_list_item_1** est un layout prédéfini qui contient un TextView pour chaque élément du tableau
- Le tableau de chaînes de caractères représentant les données à insérer dans la liste.

L'adaptateur créé sera ensuite associé à la AdapterView correspondante, en utilisant la méthode setAdapter.
Il est possible de représenter autre chose qu'un TextView pour un élément de la liste.
Il suffit pour cela d'étendre la classe ArrayAdapter et de surcharger la méthode getView. Un exemple expliquant cela sera représenté plus loin dans l'énoncé.

Une autre méthode est également possible pour faciliter le travail avec les adaptateurs.
Vous pourrez définir votre activité comme étant une ListActivity au lieu d'une Activity simple. Elle pourra ainsi implémenter toutes les fonctions de manipulation des listes directement. Pour cela, suivre les étapes suivantes :

- Dans le fichier layout XML, insérer un objet ListView, dont l'identifiant est :

```
@android :id/list
```

- Dans le code de l'activité :
  - Étendre la classe ListActivity au lieu de Activity.
  - Utiliser la méthode setListAdapter au lieu de setAdapter pour associer un adaptateur à la liste.

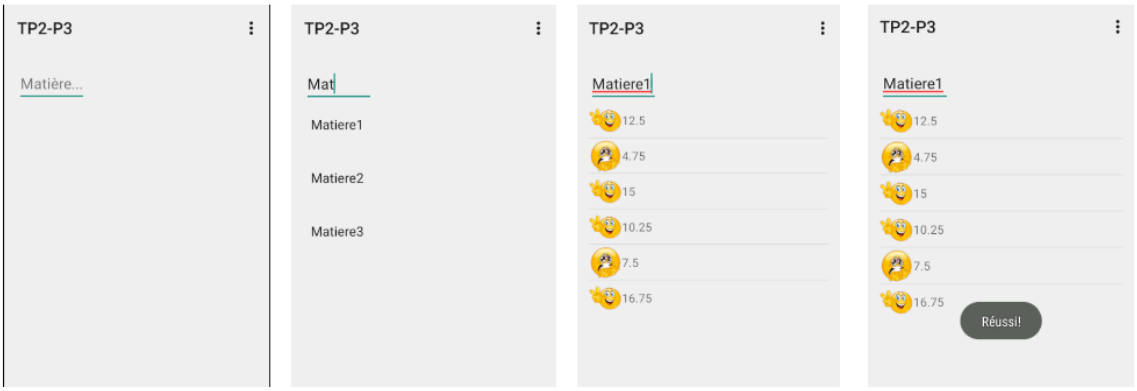
2. SimpleCursorAdapter

Utiliser cet adaptateur quand les données sont issues d'un Cursor. Cette interface fournit un accès aléatoire en lecture/écriture au résultat fourni par une requête sur une base de données.
Nous n'aborderons pas cet adaptateur dans ce document. Pour plus d'informations, consulter la documentation officielle d'Android : ici [http://developer.android.com/reference/android/widget/SimpleCursorAdapter.html]

II. Exercice : Adaptateurs et Listes

1. Objectif

L'objectif ce TP est de réaliser une application simple qui affiche les notes de plusieurs matières. On utilisera pour cela trois concepts (que nous présenterons pas à pas dans la suite) : la listView, auto-complete textView et la liste personnalisée. Le résultat qu'on désire avoir est le suivant :



2. ListView

La première étape consiste à créer une listView avec un contenu statique. Pour cela :

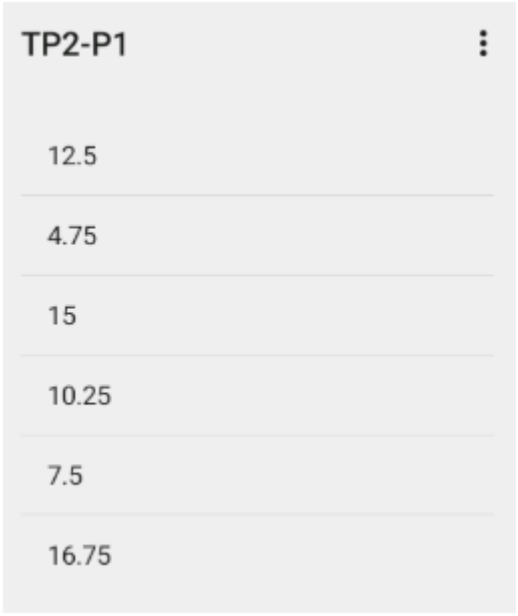
1. Créer un projet TP2, contenant une activité qu'on appellera Notes.
2. Insérer dans l'interface une widget ListView. Vous lui affecterez l'id : notesList.
3. Implémenter votre activité. Pour cela, écrire le code suivant :

```
ListView listNotes ;
   @Override
   protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);
       setContentView(R.layout.activity_main);

       String[] notes = new String[]{"10","12","13","9.5","4.5","18"
       ArrayAdapter<String> adapter = new ArrayAdapter<String>(this,
          android.R.layout.simple_list_item_1,notes);
       listNotes = findViewById(R.id.noteList) ;

       listNotes.setAdapter(adapter);
   }
```

Le résultat que vous allez obtenir doit ressembler au suivant :



Il est possible de gérer les clics sur un élement de la liste. Pour cela il y a deux principaux types de clics :

1. Le clic simple :
   Il faut surcharger la méthode *onItemClick*, comme suit :

```
listNotes.setOnItemClickListener(new AdapterView.OnItemClickListe
            @Override
            public void onItemClick(AdapterView<?> adapterView,
                  int i, long l) {
                  // Comportement au clic sur l'élément
            }
      });
```

   Les paramètres de la méthode *onItemClick* sont :

   - **adapterView :** L'AdapterView qui contient la vue sur laquelle le clic a été effectué. a été effectué.
   - **view :** La vue.(l'item)
   - **i :** La position de la vue dans la liste.
   - **id :** L'identifiant de la vue.

2. Le clic prolongé :
   Il faut surcharger la méthde *onItemLongClick*, comme suit :

```
listNotes.setOnItemLongClickListener(new AdapterView.OnItemLongClick
        @Override
        public boolean onItemLongClick(AdapterView<?> adapterVie
                int i, long l) {
            return false;
        }
    });
```

Activité 1
Créer cette première partie de l'application. Au clic sur un élément de la liste, afficher dans un toast « Réussi ! » si la note est supérieure à 10, et « Échoué.. » sinon.

3. Auto-Complete TextView

Il est parfois utile, pour faciliter la saisie de données, de fournir à l'utilisateur des propositions suite à un début de saisie dans un champs. Pour cela, un widget particulier est fourni, appelé *AutoCompleteTextView*.
Cet élément est une sous-classe de EditText, on peut donc la paramétrer de la même manière, mis à part un attribut supplémentaire : android :completionThreshold, qui indique le nombre minimum de caractères qu'un utilisateur doit entrer pour que les suggestions apparaissent.
Pour l'utiliser, suivre les étapes suivantes :

1. Dans votre layout, insérer un AutoCompleteTextView au dessus de votre liste
   notesList. Vous trouverez ce widget dans la catégorie Expert de votre palette. On
   l'appellera matièresTV. Indiquer comme completionThreshold : 3. Le code obtenu
   dans votre layout XML sera comme suit :

```
<AutoCompleteTextView
        android:layout_width="match_parent"
```

```
        android:layout_height="wrap_content"
        android:hint="matière... "
        android:id="@+id/matieresTV"
        android:completionThreshold="3"/>
```

2. Dans votre activity, indiquer dans un tableau la liste des matières qui vous seront proposées comme suggestions :

```
String[] mesMatieres = new String[] {"matière 1", "matière2","m
```

3. Associer un adaptateur à ce widget, de type ArrayAdapter. Noter que le layout utilisé comme type pour un élément de la liste est un **simple_dropdown_item_1line**.

```
matieresTV = findViewById(R.id.matieresTV);

ArrayAdapter<String> adapterMatieres = new ArrayAdapter<String>(
      android.R.layout.simple_dropdown_item_1line,mesMatieres);

matieresTV.setAdapter(adapterMatieres);
```

Le résultat obtenu aura l'allure suivante :

Pour déterminer la comportement au choix d'un élément de la liste, implémenter, comme pour la listeView, la méthode *onItemClick*.

Activité 2
Implémenter un AutoCompleteTextView comme indiqué précédemment. Définir son comportement de manière à ce que les notes affichées dans la liste changent selon la matière choisie.

4. Liste Personnalisée

Dans le cas où on aimerait que le contenu de la liste soit plus complexe qu'un simple
TextView, et qu'il puisse changer dynamiquement selon l'élément à afficher, par exemple, il faut définir votre propre adaptateur. Pour cela :

1. Commencer par définir un layout représentant une ligne de la liste. Pour cela, créer

un fichier XML dans le répertoire res/layout appelé **item_note.xml**, qui définit le
contenu d'une ligne, comme suit : (ne pas oublier d'insérer les images drawables
nécessaires. J'ai défini ici deux images : reussite et echec)

```
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android" ar
    android:layout_height="match_parent"
    android:orientation="horizontal">

    <ImageView
         android:layout_width="45dp"
         android:layout_height="45dp"
         android:id="@+id/imageNote"/>

    <TextView
        android:layout_width="match_parent"
        android:text="note"
        android:id="@+id/note"
        android:layout_height="wrap_content" />
</LinearLayout>
```

2. Créer une classe qui étend ArrayAdapter, et redéfinir la fonction getView. Pour
cela, plusieurs manières, certaines plus gourmandes en performances que d'autres 1.

    - **Méthode 1** : Méthode Classique

La méthode classique consiste à appeler un LayoutInflater , un objet permettant de convertir les éléments d'un fichier layout XML en un arbre d'objets de type View. La méthode getView, appelée à chaque fois qu'un élément de la liste est affiché à l'écran, permet de créer un objet à partir de cet élément. Par exemple, si la surface de votre smartphone peut afficher six élements de la liste, la méthode *getView* sera appelée sur les six lignes visibles seulement et pas sur les autres. Pour cela, créer une classe interne appelée MaLigneAdapter, étendant la classe ArrayAdapter, comme suit:

```
public class MaLigneAdapter extends ArrayAdapter<String> {

    Activity context ;
    String[] items ;

    public MaLigneAdapter(Activity context, String[] items) {
        super(context,R.layout.item_note,items);
        this.context = context ;
        this.items = items ;

    }



    @NonNull
    @Override
    public View getView(int position, @Nullable View convertView, @
        LayoutInflater infater = context.getLayoutInflater() ;
        View ligne = infater.inflate(R.layout.item_note,null) ;
        ImageView imgNote = ligne.findViewById(R.id.imageNote) ;
        TextView txtNote = ligne.findViewById(R.id.note);
        txtNote.setText(items[position]);
        float note = Float.valueOf(items[position]) ;
        if( note > 10)
            imgNote.setImageDrawable(context.getDrawable(R.drawable.
        else
            imgNote.setImageDrawable(context.getDrawable(R.drawable.

        return ligne ;
    }
}
```

En ce qui concerne les paramètres de la méthode **getView** :

- *position* est la position de l'item dans la liste (et donc dans l'adaptateur).
- *parent* est le layout auquel rattacher la vue.
- *convertView* est la vue qui represente un seul élèment de la liste.

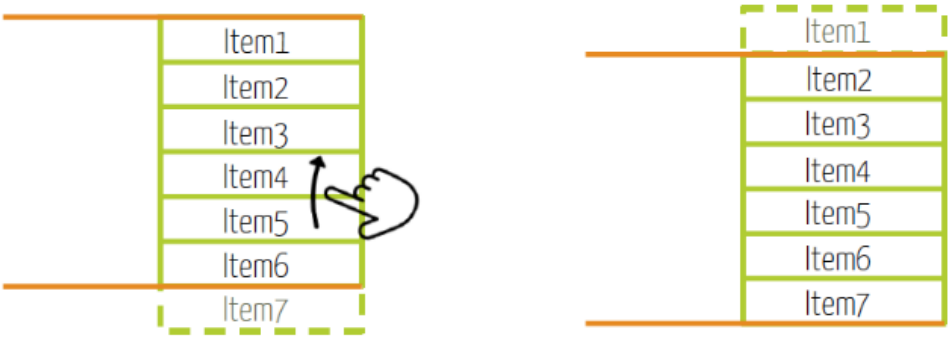
Modifier ensuite l'appel à l'adaptateur de la ListView :

```
MaLigneAdapter adapter = new MaLigneAdapter(MainActivity.this,notes)
listNotes.setAdapter(adapter);
```

- **Méthode 2** : Recycler les anciennes vues

Le problème avec la solution précédente, est que, si la liste contient un grand nombre d'éléments, la méthode getView, telle qu'elle est implémentée, créera un objet View pour chaque élément, ce qui pourra encombrer énormément la mémoire et rendre le déroulement (scroll) très lent.

Pour améliorer considérablement la performance, le concept de recyclage peut être utilisé. Considérons par exemple le cas où votre écran n'affiche que 6 éléments de la liste à la fois Pour afficher l'élément Item7, la méthode précédente créait simplement un nouvel objet et l'ajoutait à la liste. Avec le recycler, ce qu'on peut faire c'est de recycler l'objet Item1 et le réutiliser pour créer l'objet Item7. On aura ainsi à tous les instants uniquement 6 objets en mémoire, ce qui améliorera considérablement le temps de déroulement et la réactivité de l'application.



Pour faire cela, modifier le code de la méthode getView de la classe MaLigneAdapter comme suit:

```
@NonNull
@Override
public View getView(int position, @Nullable View convertView, @Non
```

```
    LayoutInflater infater = context.getLayoutInflater() ;
    if( convertView == null)
        convertView = infater.inflate(R.layout.item_note,null) ;


    ImageView imgNote = convertView.findViewById(R.id.imageNote) ;
    TextView txtNote = convertView.findViewById(R.id.note);
    txtNote.setText(items[position]);
    float note = Float.valueOf(items[position]) ;
    if( note > 10)
        imgNote.setImageDrawable(context.getDrawable(R.drawable.succ
    else
        imgNote.setImageDrawable(context.getDrawable(R.drawable.fail

    return convertView;
}
```

En effet, convertView est null uniquement au démarrage, c'est à dire pendant l'affichage des premières élements de la liste qui remplissent la surface de l'ecran(les pemières créations de vues). Mais, dès qu'on fait défiler la liste jusqu'à afficher un autre élément, convertView n'est plus null.

- **Méthode 3** : Utilisation du ViewHolder
  Un ViewHolder permet de minimiser la quantité de travail dans la fonction getView de l'adaptateur.
  Même en réutilisant l'objet convertView, vous remarquez que vous êtes entrain de refaire le même travail à plusieurs reprises : appel de la méthode findViewById pour chaque élément de la liste.
  Pour éviter de faire cela, le ViewHolder permet de sauvegarder les données des éléments passés à la ListView.
  Une classe ViewHolder aura l'allure suivante :

```
    static class ViewHolder
    {
        TextView txtNote ;
        ImageView imgNote ;

    }
```

Le principe reste semblable à la deuxième méthode : on réutilise le convertView. À la
première utilisation, on initialise le convertView, en appelant la méthode inflate, et
nous créons également une instance de la classe ViewHolder. Elle permettra d'initialiser le contenu d'un item de la liste une première fois, puis de sauvegarder cette
valeur dans un tag, attaché à l'objet convertView. Un tag représente tout type d'objet
qui peut être associé à une vue, et peut être utilisé dans votre application pour sauvegarder toute information jugée utile.

Si le convertView n'est pas null. tout ce que vous aurez à faire, est de modifier le contenu des éléments TextView et ImageView, sans avoir à les créer et initialiser à chaque appel. Le code de la fonction getView sera alors comme suit :

```
@NonNull
   @Override
   public View getView(int position, @Nullable View convertView, @N
       LayoutInflater infater = context.getLayoutInflater() ;
       ViewHolder holder = null;
       if( convertView == null)
       {
           convertView = infater.inflate(R.layout.item_note, null);
           holder = new ViewHolder() ;
           holder.txtNote = convertView.findViewById(R.id.note);
           holder.imgNote = convertView.findViewById(R.id.imageNote)

           convertView.setTag(holder);

       }
       else
       {
           holder = (ViewHolder) convertView.getTag();
       }

      holder.txtNote.setText(items[position]);
       float note = Float.valueOf(items[position]) ;
       if( note >= 10)
          holder.imgNote.setImageDrawable(context.getDrawable(R.draw
```

```
        else
            holder.imgNote.setImageDrawable(context.getDrawable(R.draw

        return convertView;
    }
```

Activité 3
Terminer la réalisation de l'exercice en utilisant respectivement les trois méthodes présentées ci-dessus.

- **Méthode 4** : RecyclerView

À partir de la version Lollipop de Android, deux nouveaux widgets ont été ajoutés à l'API : *RecyclerView* et *CardView*. Ils permettent de pallier les problèmes de performance posés par les ListViews classiques, et de proposer une vue plus ergonomique des éléments affichés sous la forme d'une liste ou d'une grille.

RecyclerView est une amélioration majeure par rapport à ListView. Il contient de nombreuses nouvelles fonctionnalités telles que *ViewHolder*, *ItemDecorator*, *LayoutManager* et *SmoothScroller*. Mais une chose qui lui donne certainement un avantage sur le ListView est la possibilité d'avoir des animations lors de l'ajout ou de la suppression d'un élément.

Quelques avantages du RecyclerView par rapport au listView :

1. Le patron ViewHolder
   Dans ListView, la définition du ViewHolder est une approche optionnelle pour améliorer la performance.
   Dans RecylerView le problème de la performance est resolu par l'utilisation la classe RecyclerView.ViewHolder. C'est l'une des principales différences entre RecyclerView et ListView. Lors de l'implémentation de RecyclerView, cette classe est utilisée pour définir un objet ViewHolder utilisé par l'adaptateur pour lier ViewHolder à une position. Un autre point à noter ici, est que lors de l'implémentation de l'adaptateur pour RecyclerView, la création d'un ViewHolder est **obligatoire**.
2. Layout Manager

Dans listView, un seul type de ListView est disponible, à savoir le ListView vertical. Vous ne pouvez pas implémenter un ListView avec un défilement horizontal (il existe des moyens pour l'implémenter mais un peu compliqués).
Dans RecyclerView nous avons la possibiliter d'implementer des listes horizontales. En fait, il prend en charge plusieurs types de listes. Pour prendre en charge plusieurs types de listes, il utilise la classe *RecyclerView.LayoutManager*. RecyclerView prend en charge trois types de *LayoutManager* prédéfinis:

- LinearLayoutManager : C'est le *LayoutManager* le plus utilisé. Grâce à cela, vous pouvez créer des listes horizontales et verticales. cela, vous pouvez créer des listes horizontales et verticales.
- StaggeredGridLayoutManager : Utilisé pour créer des listes décalées comme suit : comme suit :

- GridLayoutManager : Peut être utilisé pour afficher des grilles.

3. ItemAnimator
   RecyclerView fournit la classe RecyclerView.ItemAnimator pour gérer les

animations. Grâce à cette classe, des animations personnalisées peuvent être définies pour l'ajout, la suppression et le déplacement d'événements. En outre, il fournit un *DefaultItemAnimator*, au cas où vous n'avez pas besoin de personnalisations.

4. OnItemTouchListener
   ListView a une implémentation simple pour la détection des clics, c'est-à-dire en utilisant l'interface AdapterView.OnItemClickListener.
   Cependant, l'interface RecyclerView.OnItemTouchListener est utilisée pour détecter les événements. Cela complique un peu la mise en œuvre, mais cela donne un meilleur contrôle au développeur pour intercepter les événements.

Pour plus de détails, vous pouvez consulter la documentation officielle ici [https://developer.android.com/reference/android/support/v7/widget/RecyclerView.html].

Homework

Pour appliquer les nouvelles notions de *RecyclerView* de *CardView*, vous devez faire une petite application de gestion des contacts téléphoniques.
Cette application aura comme principale interface une liste de contacts, un contact est caractérisé par son image, son nom, son prénom, son adresse email et son numéro de téléphone (Pensez donc à créer une liste d'objets Contact).
L'utilisateur aura la possiblité de chercher un contact (utiliser un moyen que vous jugez utile) et de lui contacter à travers son email ou son numéro de téléphone.

N’oubliez pas de respecter les principes du Material Design!