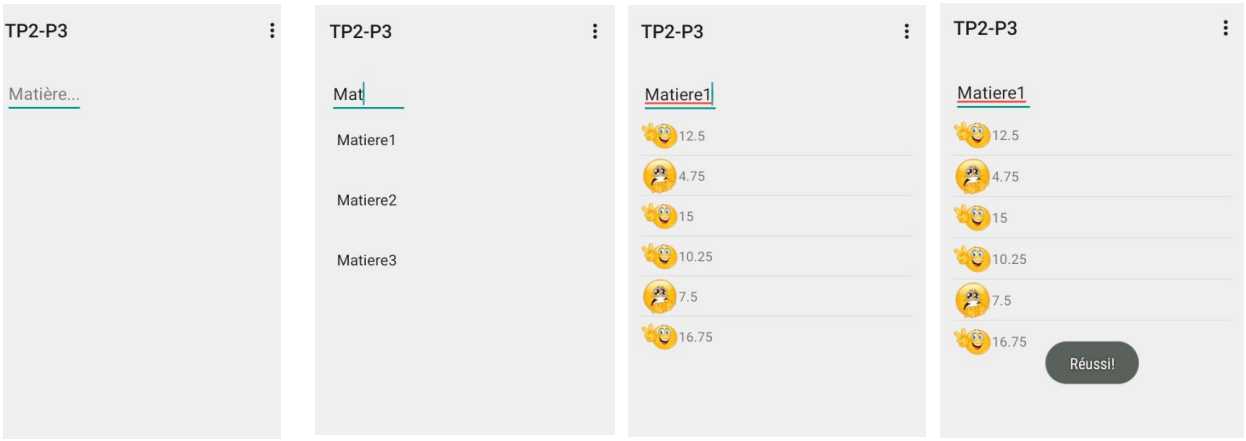
Adaptateurs et listes

Objectif :

L'objectif de ce TP est de réaliser une application simple qui affiche les notes de plusieurs matières, on utilisera pour cela trois concepts : la ListView, auto-complete TextView et la liste personnalisée.

Le résultat qu'on désire avoir est le suivant :



I : ListView

La première étape consiste à créer une ListView avec un contenu statique, pour cela :

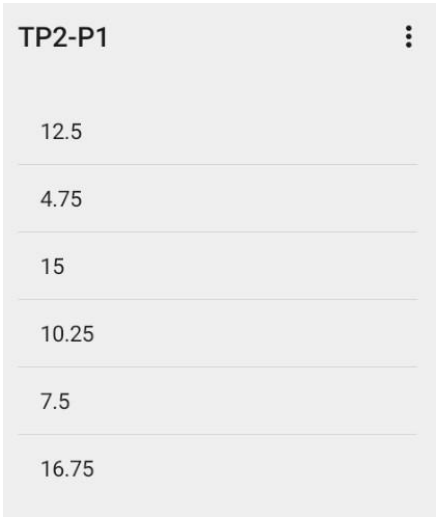
- Créer un projet TP2, contenant une activité qu'on appellera Notes
- Insérer dans l'interface un widget ListView ayant l'identifian : notesList
- Implémentez votre activité, pour cela écrire le code suivant :

```
ListView notesList;
String[] mesNotes = {"12.5", "4.75", "15", "10.25", "7.5", "16.75"};

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_notes);

    notesList = (ListView) findViewById(R.id.notesList);
    ArrayAdapter<String> listAdapter = new ArrayAdapter<String>(
            this, android.R.layout.simple_list_item_1, mesNotes);
    notesList.setAdapter(listAdapter);
}
```

Le résultat que vous allez obtenir doit ressembler au suivant :



Pour ajouter un comportement au clic sur un élément de la liste, il faut surcharger la méthode **onItemClick** comme suit :

```
notesList.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
        // behaviour when you click on the item "view"
    }
});
```

TAF : 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 "Echoué" sinon.

II. Auto-complete TextView

Il est parfois utile, pour faciliter la saisie des 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 apparaîssent.

Pour l'utiliser, suivez les étapes suivantes :

- Dans votre activité, insérez un AutoCompleteTextView (id = matiereTV) au dessus de votre liste notesList
- Indiquez comme completionTreshold :3

```xml
<AutoCompleteTextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:hint="Matière..."
    android:id="@+id/matieresTV"
    android:completionThreshold="3"
     />
```

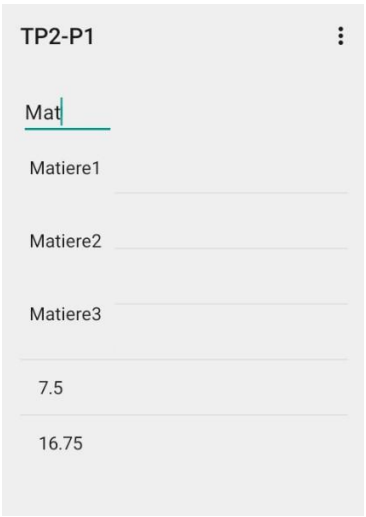
- Dans votre activité, indiquez dans un tableau la liste des matières qui vous proposés comme suggestions :

```java
String[] mesMatieres = {"Matiere1", "Matiere2", "Matiere3"};
```

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

```java
matieresTV = (AutoCompleteTextView) findViewById(R.id.matieresTV);
matieresTV.setAdapter(new ArrayAdapter<String>(
        this, android.R.layout.simple_dropdown_item_1line, mesMatieres));
```

Le résultat obtenu aura l'allure suivante



Pour déterminer le comportement au choix d'un élément de la liste, implémentez la méthode **onItemClick** :

```java
matieresTV.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
        //behaviour when an item "view" is clicked
    }
});
```

TAF : Implémentez un AutoCompleteTextView comme indiqué précédemment, définir son comportement de manière à ce que les notes affichées dans la liste changent selon la matière choisie

III : 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, il faut définir son propre adaptateur. Pour cela :

- Commencez par définir un layout représentant une ligne de la liste. Pour cela créez un fichier xml dans le répertoire layout appelé ma_ligne.xml, qui définit le contenu d'une ligne

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="horizontal" android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="40dp"
        android:layout_height="42dp"
        android:id="@+id/monImage"
        android:src="@drawable/reussite"
        />
    <TextView
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:text="Note.."
        android:id="@+id/note"
        android:gravity="center"/>

</LinearLayout>
```

- Créez une classe qui étend ArrayAdapter, et redéfinir la méthode getView. Pour cela plusieurs méthodes existent :

a. Méthode 1 : méthode classique

Cette méthode consiste à appeler un layoutInflater, un objet qui permet de convertir les éléments d'un fichier layout XML en un arbre d'objets de type View. La méthode getView, appelée) chaque fois qu'un élément de la liste est affiché à l'écran, permet de créer un objet à partir de cet élément. Créez une classe appelée MaLigneAdapter étendant ArrayAdapter comme suit :

```
class MaLigneAdapter extends ArrayAdapter<String>{
    Activity context;
    String[] items;
    MaLigneAdapter(Activity context, String[] items){
        super(context, R.layout.ma_ligne,items);
        this.context = context;
        this.items = items;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View ligne = inflater.inflate(R.layout.ma_ligne,null);
        TextView label = (TextView) ligne.findViewById(R.id.note);
        ImageView icone = (ImageView) ligne.findViewById(R.id.monImage);
        label.setText(items[position]);
        float note = Float.valueOf(items[position]);
        icone.setImageResource(
                (note >= 10)? R.drawable.reussite : R.drawable.echec
        );

        return ligne;
    }

}
```

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

```
notesList.setAdapter(new MaLigneAdapter(this, new String[0]));
```

b. 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 7, 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 1 et le réutiliser pour créer l'objet 7. On aura ainsi à tous les instants uniquement 6 objets en mémoire, ce qui améliora 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 :

```java
@Override
public View getView(int position, View convertView, ViewGroup parent) {
    LayoutInflater inflater = context.getLayoutInflater();
    if (convertView == null){
        convertView = inflater.inflate(R.layout.ma_ligne,null);
    }
    TextView label = (TextView) convertView.findViewById(R.id.note);
    ImageView icone = (ImageView) convertView.findViewById(R.id.monImage);
    label.setText(items[position]);
    float note = Float.valueOf(items[position]);
    icone.setImageResource(
            (note >= 10)? R.drawable.reussite : R.drawable.echec
    );
    return convertView;

}
```