CS425 - Web and Mobile Software Engineering
Dr. Lilia Sfaxi
17 avril 2017

MedTech

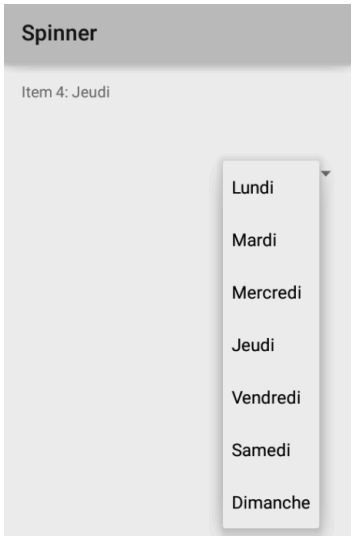
Advanced Graphical Elements: Adapters and Lists

Android Lab 02

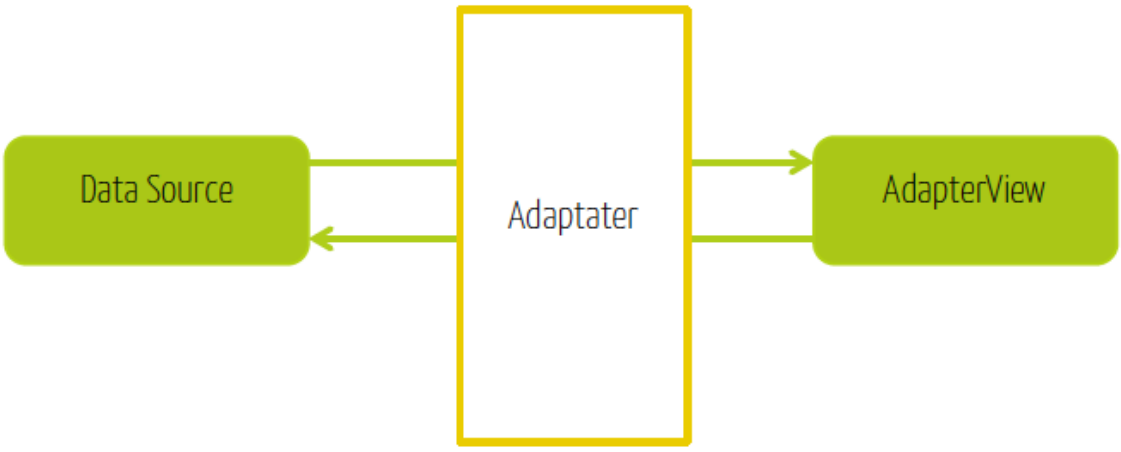
I. Adapters

An adapter object acts as a bridge between a view (called an AdapterView) and the underlying data. An AdapterView is a complex view that contains other views, and that needs an adapter to construct its child views.

The most common AdapterViews are: ListView, GridView and Spinner.



An adapter enables to transform the data of a DataSource (an array or a cursor) to widgets to insert them in an AdapterView.



The getView() method of the Adapter class returns, for each element of the AdapterView, a View object. This method is called each time an element of the list is displayed in the screen.

There are mainly two types of adapters: ArrayAdapter and SimpleCursorAdapter.

1. ArrayAdapter

Use this adapter when your data source is an array. By default, this adapter creates a view for each element of the array, calls the *toString* method for each element, and places its content in a TextView object.

An ArrayAdapter is created as follows.

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

The constructor's arguments are:

- The context of the application: usually the activity where the AdapterView is created.
- A layout representing the type of a list element: the default layout (android.R.layout.simple_list_item_1) is a predefined layout with a TextView for each element of the array.
- The source array containing the data to be inserted in the View.

The adapter is then associated to a corresponding AdapterView, using the *setAdapter* method.

It is possible to represent a list element other than a TextView. To do this, you have to extend the ArrayAdapter class, and override the getView method. This will be explained later.

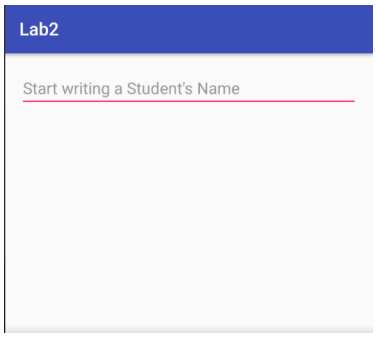
2. SimpleCursorAdapter

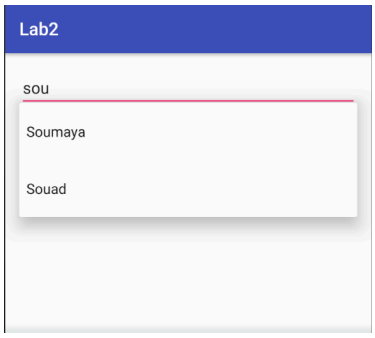
Use this adapter when the data are issued from a Cursor. This interface provides a read/write random access to a resultset of a database request.

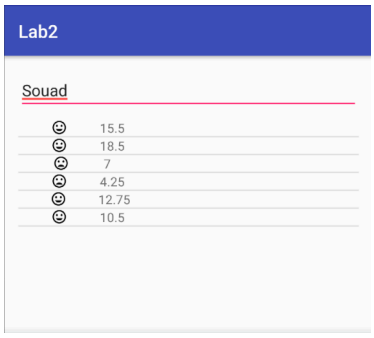
II. Exercise: Adapters and Lists

1. Objective

The goal of this lab is to create a simple application that displays the grades of a set of students. We will use for this three concepts, that we will present one by one in the rest of the document: a list view, an auto-complete text view and a custom list. The desired result is:







2. ListView

The first step consists in creating a listView with a static content:

1. Create a project Lab2, containing an activity called GradesActivity.
2. Insert in the layout a ListView widget, which id is : gradesList
3. Write your activity's code:

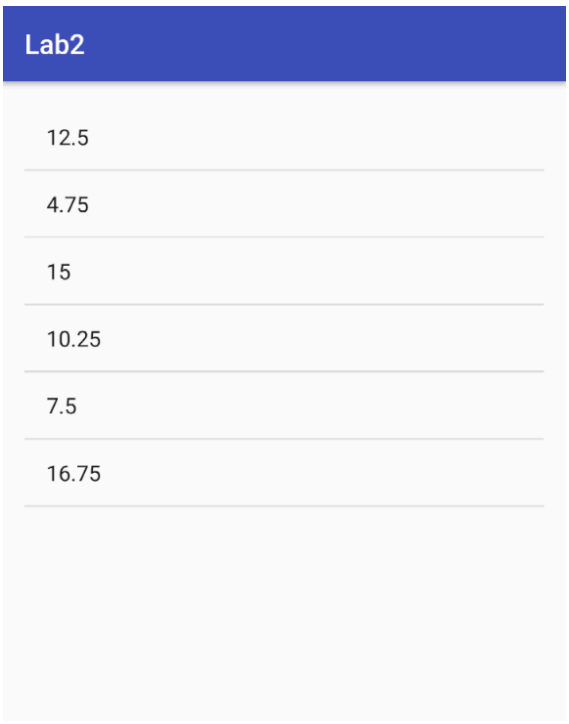
```
ListView gradesList;
String[] myGrades = {"12.5", "4.75", "15", "10.25", "7.5", "16.75"};

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

    gradesList = (ListView)findViewById(R.id.gradesList);


    ArrayAdapter<String> adapter = new ArrayAdapter<String>(
            this, android.R.layout.simple_list_item_1, myGrades);
    gradesList.setAdapter(adapter);
}
```

The result should look like the following:



To add a behaviour to the click on a list element, override the onItemClick method, as follows:

```
gradesList.setOnItemClickListener(new AdapterView.OnItemClickListener() {
    @Override
    public void onItemClick(AdapterView<?> adapterView, View view, int i, long l) {
        // behaviour when clicking on a list item
    }
});
```

Activity 1: Create a first version of the application as described above. A click on a list item will display « Pass! » if the grade is above 10, and « Fail! » else.

3. Auto-Complete TextView

It is sometimes useful, to make writing data to the device easier, to use some completion propositions. To do this, an AutoCompleteTextView is provided.

This element is a sub-class of EditText, that we can configure and use the same way as this latter, with an additional attribute: *android:completionThreshold*, representing the number of characters that the user has to write before the suggestions appear.

To use it, follow these instructions:

1. In your layout, insert an AutoCompleteTextView above the list, with the id *students,* and a completionThreshold of 3.

```
<AutoCompleteTextView
    android:completionThreshold="3"
    android:id="@+id/students"
    android:hint="Start writing a Student's Name"
```

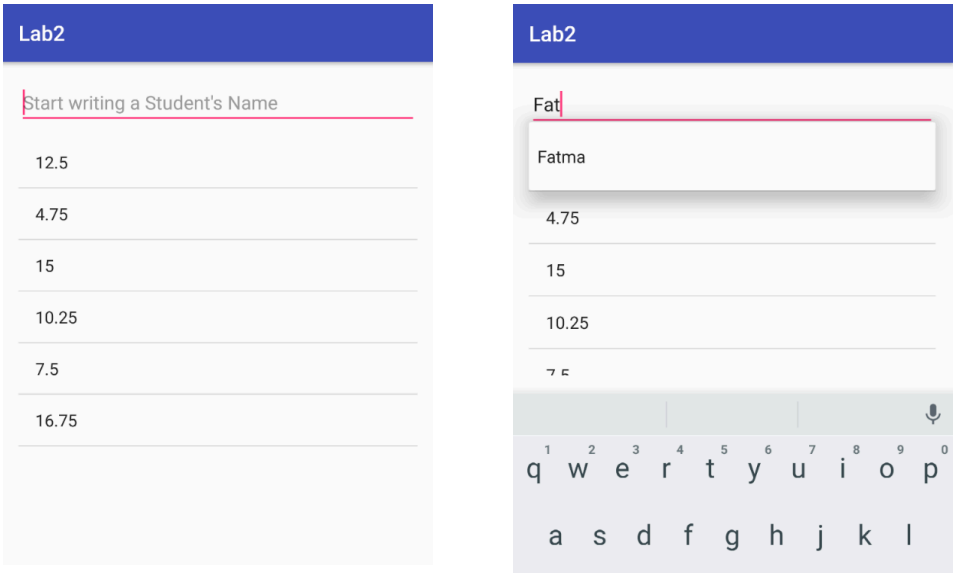
2. In your activity, create an array with the names of the students, that will represent the suggestions.

```
String[] myStudents = {"Chayma","Fatma", "Omar", "Soumaya"};
```

3. Associate an adapter to this widget. The layout used here for an element of the list is: simple_dropdown_item1_line

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

The result looks like the following:



To define the behaviour when an element is chosen, implement the method onItemClick:

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

Activity 2: Add the AutoCompleteTextView to your previous activity. Define its behaviour in a way that, when the selected student name changes, the displayed grades change.

4. Custom List

In the case where we want the content of the list to be more complex and dynamic than a simple textView, we have to define our own adapter.

1. Start by defining a layout representing one element of the list. To do this, create an XML layout in the res/layout directory, which defines the content of a line. You have first to add two images in the drawable directory, one for success, and one for failure.

```
<LinearLayout
    android:orientation="horizontal"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <ImageView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        app:srcCompat="@drawable/ic_mood"
        android:id="@+id/image"
        android:layout_weight="0.28" />

    <TextView
        android:text="TextView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:id="@+id/grade"
        android:layout_weight="1" />

</LinearLayout>
```

2. Crete a class extending ArrayAdapter, and redefine the function *getView*. To do this, many ways are possible, some greedier in memory than others[1]!

   - Method 1: Classical way

The classical way would be to call a LayoutInflater, an object that converts the elements in an XML layout into a tree of View objects. The *getView* method, called each time an element of the list is displayed on the screen, creates an object from this element.

To do this, create an internal class called *MyLineAdapter*, which extends the *ArrayAdapter* class.

```
class MyLineAdapter extends ArrayAdapter<String>{
    Activity context;
    String[] items;

    MyLineAdapter(Activity context, String[] items){
        super(context,R.layout.line, items);
        this.context = context;
        this.items = items;
    }

    @NonNull
    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        LayoutInflater inflater = context.getLayoutInflater();
        View line = inflater.inflate(R.layout.line, null);
        TextView label = (TextView) line.findViewById(R.id.grade);
        ImageView image = (ImageView)line.findViewById(R.id.image);
        label.setText(items[position]);
        float grade = Float.valueOf(items[position]);
        image.setImageResource((grade >= 10)?R.drawable.ic_mood: R.drawable.ic_mood_bad);

        return line;
    }
}
```

[1] [1] See the 10 first minutes of the Google I/O 2009 conference: Make your Android UI Fast and Efficient : https://www.youtube.com/watch?v=N6YdwzAvwOA

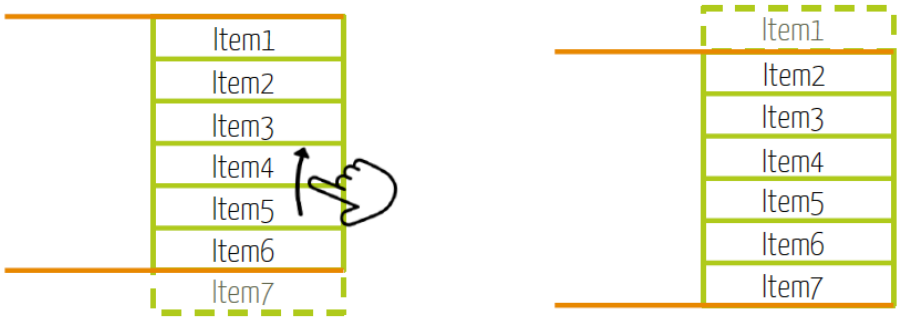
Change then the call to the adapter of the list view.

```
MyLineAdapter adapter = new MyLineAdapter(GradesActivity.this,allGrades.get(selectedStudent));
gradesList.setAdapter(adapter);
```

- Method 2: Recycle old views

The problem with the first solution is that, if the list contains a big number of elements, the *getView* method will create a new View object for each one, a fact that will congest the memory and make the scroll really slow.

To improve the performance, the recycling concept is used. Let's consider for example the case where the screen can only display 6 elements at once.



To display a 7th element, the previous method will simply create a new object and add it to the list. With the recycler, what we can do is to recycle the object Item1, that no longer appears on the screen, and reuse it for Item7. We will thus have at all times only 6 objects in memory.

To do this, change the code of the *getView* method of your class *MyLineAdapter* as follows:

```
public View getView(int position, View convertView, ViewGroup parent) {
    LayoutInflater inflater = context.getLayoutInflater();
    if (convertView == null){
        convertView = inflater.inflate(R.layout.line, null);
    }

    TextView label = (TextView) convertView.findViewById(R.id.grade);
    ImageView image = (ImageView)convertView.findViewById(R.id.image);
    label.setText(items[position]);
    float grade = Float.valueOf(items[position]);
    image.setImageResource((grade >= 10)?R.drawable.ic_mood: R.drawable.ic_mood_bad);

    return convertView;
}
```

- Method 3: Use the ViewHolder

A ViewHolder can help minimise the work for the getView method. Even when reusing the convertView, the method calls findViewById for each new element. To avoid this, a View

Holder can save the data of the elements of the ListView. To do this, create a ViewHolder class as follows:

```
static class ViewHolder{
    TextView label;
    ImageView image;
}
```

The same principle as the previous method stands: the convertView is reused. But at the first use, the convertView is initialised with the inflate method, and we also create an instance of the ViewHolder. It will enable to initialise the content of an item a first time, then to save this value in a tag attached to the convertView. A tag represents any object that can be associated to a view, and can be used in the application to save any needed information.

If the convertView is not null, all you have to do is to modify the content of the TextView and ImageView elements, without creating them again. The getView method will become as follows:

```
public View getView(int position, View convertView, ViewGroup parent) {
    LayoutInflater inflater = context.getLayoutInflater();
    ViewHolder holder;
    if (convertView == null){
        convertView = inflater.inflate(R.layout.line, null);

        holder = new ViewHolder();
        holder.label = (TextView) convertView.findViewById(R.id.grade);
        holder.image = (ImageView)convertView.findViewById(R.id.image);

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

    holder.label.setText(items[position]);
    float grade = Float.valueOf(items[position]);
    holder.image.setImageResource((grade >= 10)?R.drawable.ic_mood: R.drawable.ic_mood_bad);

    return convertView;
}
```

- Method 4: RecyclerView

Since the Lollipop version of Android, two new widgets were added to the API: RecyclerView and CardView. They help preventing the performance problems of the ListView by embedding the ViewHolder pattern implicitly, and use more ergonomic elements as a display.

Activity 3: Continue the implementation of the example using respectively the three first methods.