

Cours développement d'application mobile native

Interaction avec L'utilisateur

Spinner
ListView
RecyclerView

Hend Ben Ayed kharrat







2023-2024

Ordre du Jour

- Gestion des listes de données
- Spinner
- RecyclerView

Gestion des listes de données

Construire une ListView

ListViews	
	bike
	boat
	bus
	car
	railway
	run

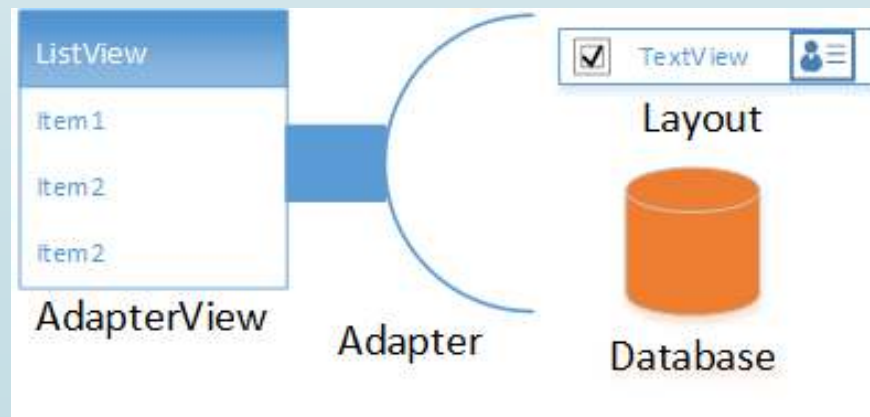


GESTION DES LISTES DE DONNÉES

4

La gestion des listes se divise en deux parties distinctes.

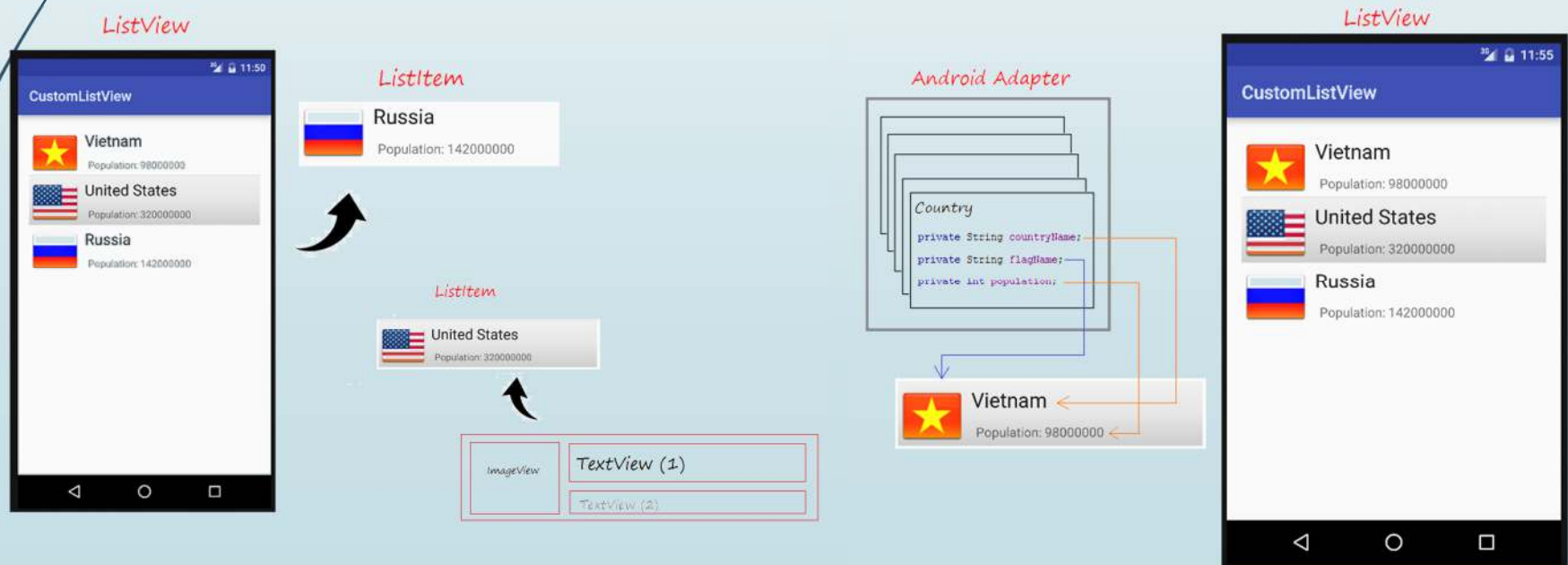
- Les **Adapter** qui sont les objets qui gèrent les données, mais pas leur affichage ou leur comportement en cas d'interaction avec l'utilisateur. On peut considérer un adaptateur comme une intermédiaire entre les données et la vue qui représente ces données.
- Les **AdapterView**, qui vont gérer l'affichage et l'interaction avec l'utilisateur, mais sur lesquels on ne peut pas effectuer d'opération de modification des données.



GESTION DES LISTES DE DONNÉES

5

- Le comportement typique pour afficher une liste depuis un ensemble de données est celui-ci :
 - On donne **à l'Adapter** une liste d'éléments à traiter et la manière dont ils doivent l'être,
 - On passe cet adaptateur à un AdapterView.
 - Dans **l'AdapterView**, l'adaptateur va créer un widget pour chaque élément en fonction des informations fournies en amont.



- Adapter est une interface qui définit les comportements généraux des adaptateurs.
- L'adapter aide à convertir les éléments de données en des views items de android
 - Si on veut construire un widget simple, on retiendra trois principaux adaptateurs :
 - **ArrayAdapter**, qui permet d'afficher les informations simples;
 - **SimpleAdapter** est quant à lui utile dès qu'il s'agit d'écrire plusieurs informations pour chaque élément (s'il y a deux textes dans l'élément par exemple) ;
 - **CursorAdapter** pour adapter le contenu qui provient d'une base de données. On y reviendra dès qu'on abordera l'accès une base de données

LES VUES RESPONSABLES DE L'AFFICHAGE DES LISTES : LES ADAPTERVIEW

7

- On trouve la classe AdapterView dans le package `android.widget.AdapterView`.
 - l'Adapter s'occupe des éléments en tant que données, alors que l'AdapterView s'occupe de les afficher et veille aux interactions avec un utilisateur (sélection d'un élément par exemple)
 - On trouve trois principaux AdapterView :
 - ListView, pour simplement afficher des éléments les uns après les autres ;
 - GridView, afin d'organiser les éléments sous la forme d'une grille ;
 - Spinner, qui est une liste déroulante.
- Pour associer un adapter à un AdapterView, on utilise la méthode :
`void setAdapter` (Adapter adapter) qui se chargera de peupler la vue.

LES ARRAYADAPTERS (LISTES SIMPLES)

8

- La classe ArrayAdapter se trouve dans le package android.widget.ArrayAdapter.
- On va considérer les constructeurs puliques suivants :

[ArrayAdapter](#)([Context](#) context, int resource)

[ArrayAdapter](#)([Context](#) context, int resource, int textViewResourceld)

[ArrayAdapter](#)([Context](#) context, int resource, T[] objects)

[ArrayAdapter](#)([Context](#) context, int resource, int
textViewResourceld, T[] objects)

[ArrayAdapter](#)([Context](#) context, int resource, [List](#)<T> objects)

[ArrayAdapter](#)([Context](#) context, int resource, int
textViewResourceld, [List](#)<T> objects)

LES SIMPLEADAPTER (LISTES PLUS COMPLEXES)

On peut utiliser la classe SimpleAdapter à partir du package `android.widget.SimpleAdapter`.

Le SimpleAdapter est utile pour afficher simplement **plusieurs informations par élément**. En réalité, pour chaque information de l'élément on aura une vue dédiée qui affichera l'information voulue. Ainsi, on peut avoir du texte, une image...

Public constructors

```
SimpleAdapter(Context context, List<? extends Map<String, ?>> data, int resource, String[] from, int[] to)
```

Relations entre la vue et les données

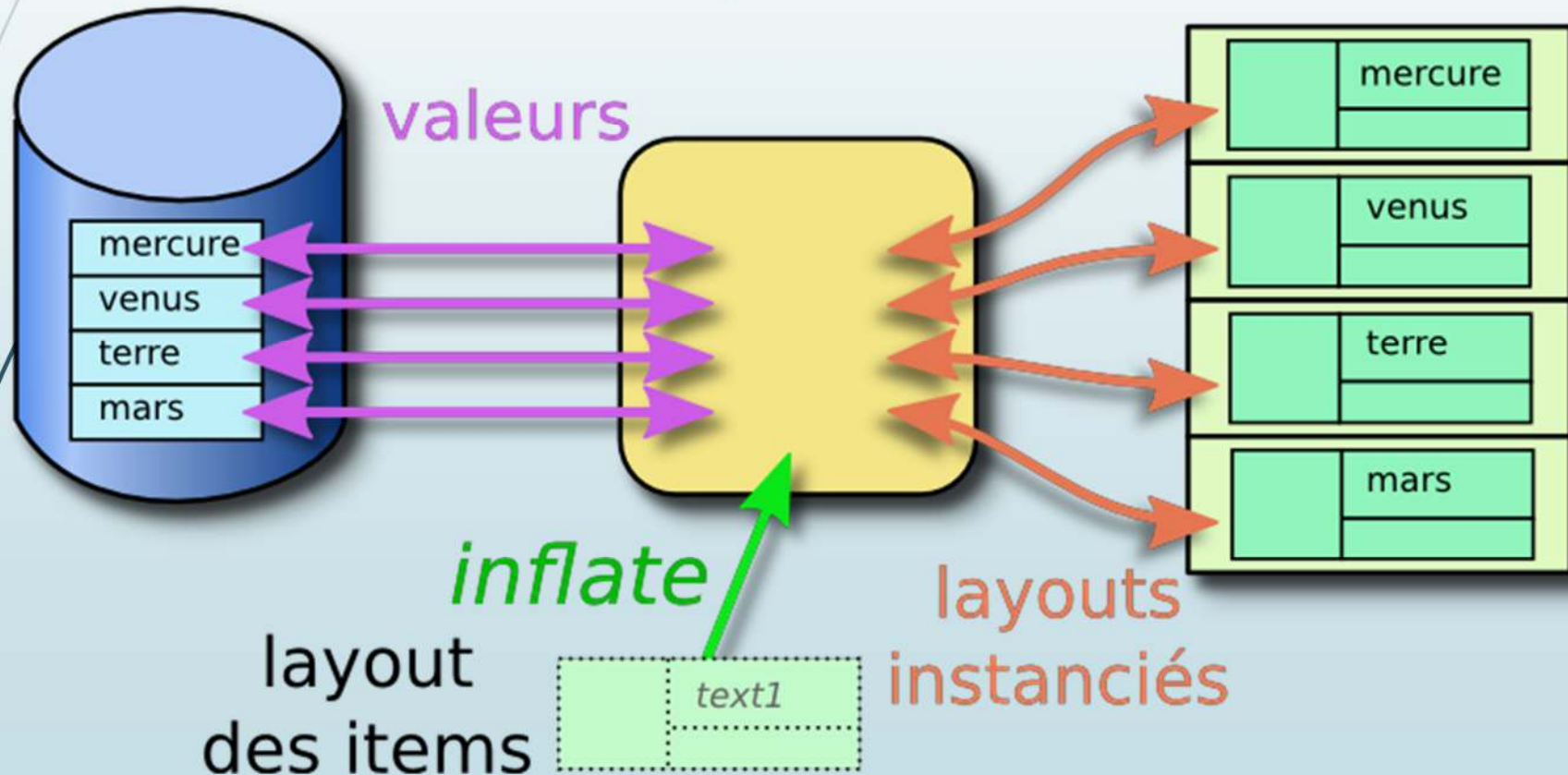
10

Un `ListView` affiche les items à l'aide d'un *adaptateur* (*adapter*).

Données

Adaptateur

Vue



ArrayAdapter<Type> pour les listes

11

Il permet d'afficher les données d'un ArrayList, mais il est limité à une seule chaîne par item, par exemple le nom d'une planète, fournie par sa méthode `toString()`. Son constructeur :

```
ArrayAdapter(Context context, int item_layout_id, int  
textView_id, List<T> données)
```

context c'est l'activité qui crée cet adaptateur, mettre `this`
item_layout_id identifiant du layout des items, p. ex.

`android.R.layout.simple_list_item_1` ou
`R.layout.item`

textView_id identifiant du TextView dans ce layout, p. ex.

`android.R.id.text1` ou `R.id.item_nom`

données c'est la liste contenant les données (List est une
surclasse de ArrayList)

Liste d'éléments cochables

12

Android offre des listes cochables comme celles-ci :

Mercure	<input checked="" type="checkbox"/>	Mercure	<input type="checkbox"/>
Vénus	<input checked="" type="checkbox"/>	Vénus	<input type="checkbox"/>
Terre	<input checked="" type="checkbox"/>	Terre	<input checked="" type="checkbox"/>
Mars	<input checked="" type="checkbox"/>	Mars	<input checked="" type="checkbox"/>
Jupiter	<input checked="" type="checkbox"/>	Jupiter	<input type="checkbox"/>

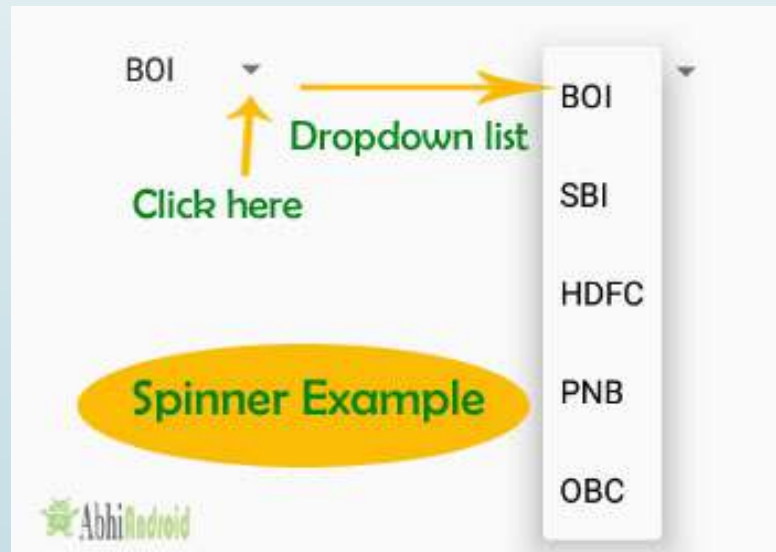
Le style de la case à cocher dépend du choix unique ou multiple.

MÉTHODES COMMUNES À TOUS LES ADAPTATEURS

13

- Pour ajouter un objet à un adaptateur, on peut utiliser la méthode:
 - void `add` (T object) ou
- l'insérer à une position particulière avec:
 - void `insert` (T object, int position).
- Pour récupérer un objet dont on connaît la position on utilise la méthode T
 - `getItem` (int position),
- et pour récupérer la position d'un objet précis on utilise la méthode int
 - `getPosition` (T object).
- Pour supprimer un objet la méthode
 - void `remove` (T object) est utilisée
- Pour vider complètement l'adaptateur
 - void `clear` ().

Construire un Spinner



La classe Spinner se trouve dans le package `android.widget.Spinner`.

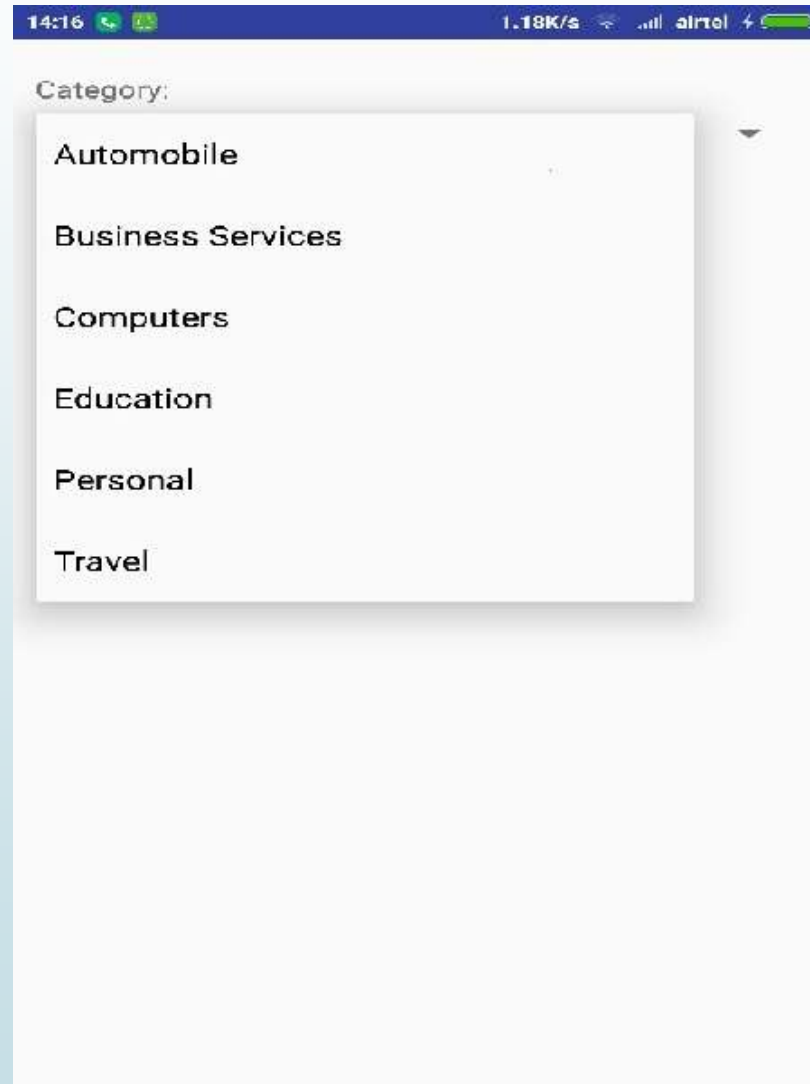
On utilisera deux vues. Une pour l'élément sélectionné qui est affiché, et une pour la liste d'éléments sélectionnables.

Suivez ces étapes:

1. Créez un Spinner élément dans votre mise en page XML et spécifiez ses valeurs à l'aide d'un tableau et d'un ArrayAdapter.
2. Créez le Spinner et son adaptateur à l'aide de la SpinnerAdapter classe.
3. Pour définir le rappel de sélection pour le Spinner, mettez à jour le Activity qui utilise le Spinner pour implémenter l' AdapterView.OnItemSelectedListener interface.

LES SPINNERS: EXEMPLE

16



LES SPINNERS: EXEMPLE

17

```
package com.example.spinner;
import java.util.ArrayList;
import java.util.List;
import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.AdapterView;
import android.widget.AdapterView.OnItemClickListener;
import android.widget.ArrayAdapter;
import android.widget.Spinner;
import android.widget.Toast;

class AndroidSpinnerExampleActivity extends Activity
implements OnItemSelectedListener{

@Override public void onCreate(Bundle savedInstanceState) {
super.onCreate(savedInstanceState);
setContentView(R.layout.main);
// Spinner element
Spinner spinner = (Spinner) findViewById(R.id.spinner);
// Spinner click listener
spinner.setOnItemSelectedListener(this);
```

```
// Spinner Drop down elements
List<String> categories = new ArrayList<String>();
categories.add("Automobile");
categories.add("Business Services");
categories.add("Computers");
categories.add("Education");
categories.add("Personal");
categories.add("Travel");
// Creating adapter for spinner
ArrayAdapter<String> dataAdapter = new
ArrayAdapter<String>(this,
android.R.layout.simple_spinner_item, categories);
// attaching data adapter to spinner
spinner.setAdapter(dataAdapter); }
```

LES SPINNERS: EXAMPLE

19

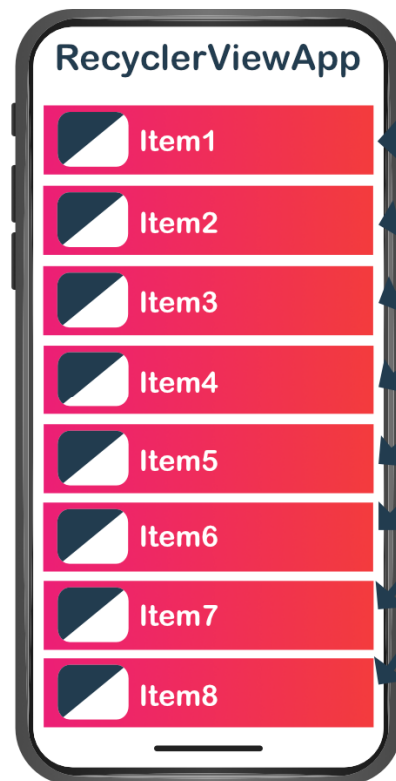
```
@Override public void onItemSelected(AdapterView<?> parent,
View view, int position, long id) {

    // On selecting a spinner item
    String item = parent.getItemAtPosition(position).toString();

    // Showing selected spinner item
    Toast.makeText(parent.getContext(), "Selected: " + item,
    Toast.LENGTH_LONG).show(); }

public void onNothingSelected(AdapterView<?> arg0) {
    // TODO Auto-generated method stub } }
```

Construire un RecyclerView



View Holders

Populates the view elements in the UI which is visible to the user

Adapter

Bind data to ViewHolders

Data Set

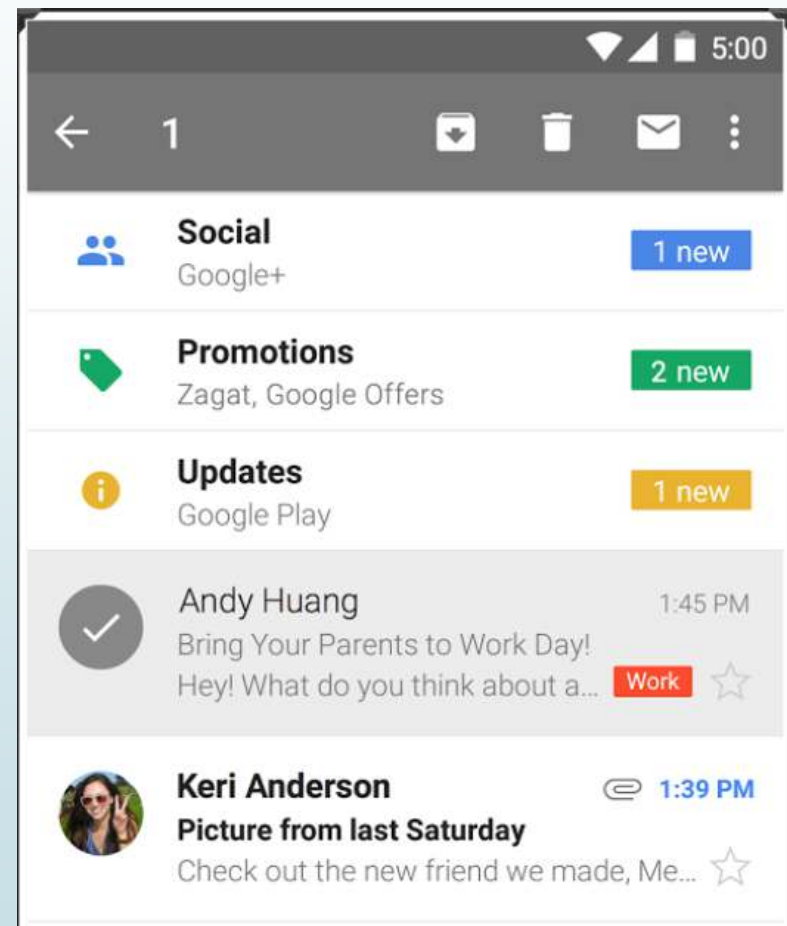
List of items to populate in the ViewHolders



QU'EST CE QU'UNE RECYCLERVIEW

21

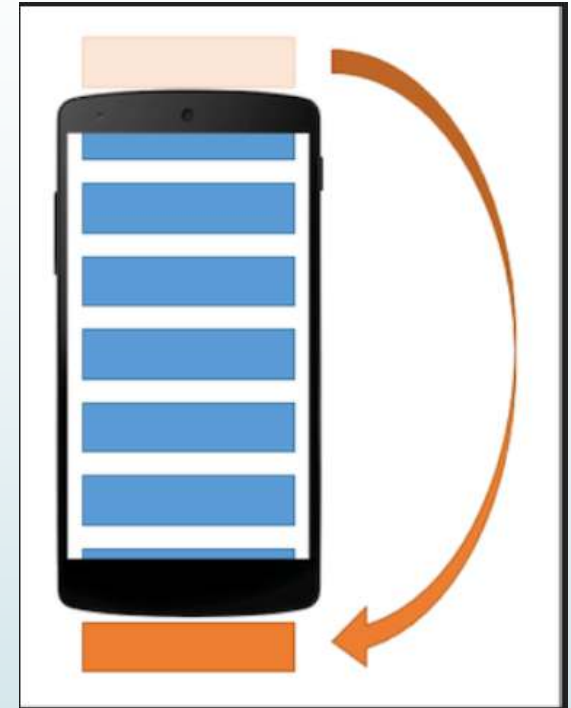
- La RecyclerView est un objet qui permet d'afficher une liste (ex.: tweets, posts, mails...). Elle remplace la ListView.
- Elle contient des avantages non négligeables et devient indispensable pour tout projet d'ampleur.



LES DIFFÉRENCES AVEC LA LISTVIEW

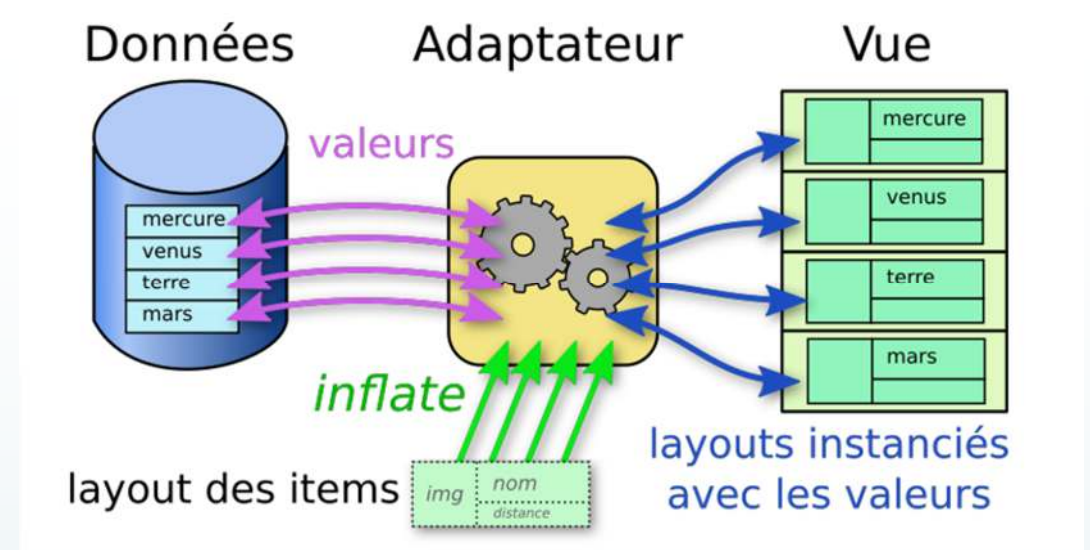
22

- ▶ Dans une ListView classique, le système se charge de créer toutes les vues qui correspondent aux éléments de votre liste avant même qu'elles ne soient visible à l'écran. Ce système peut saturer la mémoire de votre machine virtuelle et lever une erreur `OutOfMemoryError`.
- ▶ Dans une RecyclerView, le système ne va charger que les vues qui seront visible à l'écran ! Lors du scroll, elle réutilisera les vues qui disparaissent pour charger les éléments suivants.



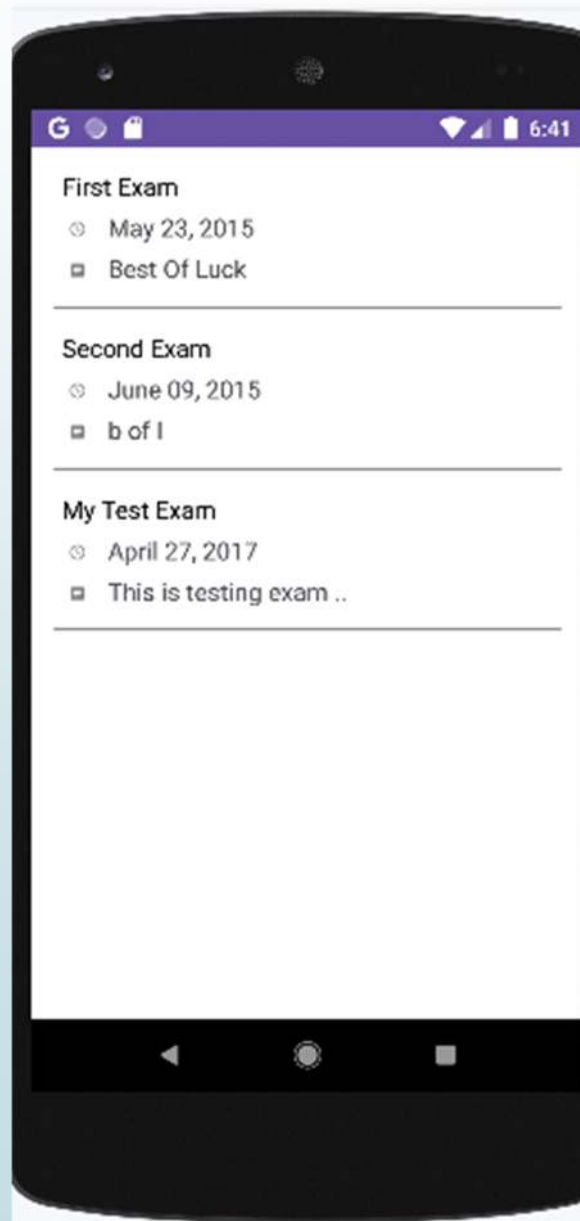
Concepts

23



- La vue ne sert qu'à afficher les éléments de la liste. En réalité, seuls quelques éléments seront visibles en même temps. Cela dépend de la hauteur de la liste et la hauteur des éléments.
- Le principe du RecyclerView est de ne gérer que les éléments visibles. Ceux qui ne sont pas visibles ne sont pas mémorisés. Mais lorsqu'on fait défiler la liste ainsi qu'au début, de nouveaux éléments doivent être rendus visibles.
- Le RecyclerView demande alors à l'adaptateur de lui instancier (*inflate*) les vues pour afficher les éléments.
- Le nom « RecyclerView » vient de l'astuce : les vues qui deviennent invisibles à cause du défilement vertical sont recyclées et renvoyées de l'autre côté mais en changeant seulement le contenu à afficher.

Exemple à construire



Nous allons dans un premier temps, créer un objet java qui va représenter l'entité qui sera affiché dans la liste, dans notre cas, un exam, nous développons donc une liste de exam

```
public class ExamClass {  
    String name;  
    String date;  
    String message;  
    ExamClass(String name, String date, String  
message)  
    {  
        this.name = name;  
        this.date = date;  
        this.message = message;  
    }  
    public String getName() {...  
    public void setName(String name) {....  
    public String getDate() {.....  
    public void setDate(String date) {.....  
    public String getMessage() {....  
    public void setMessage(String message) {.....  
}
```

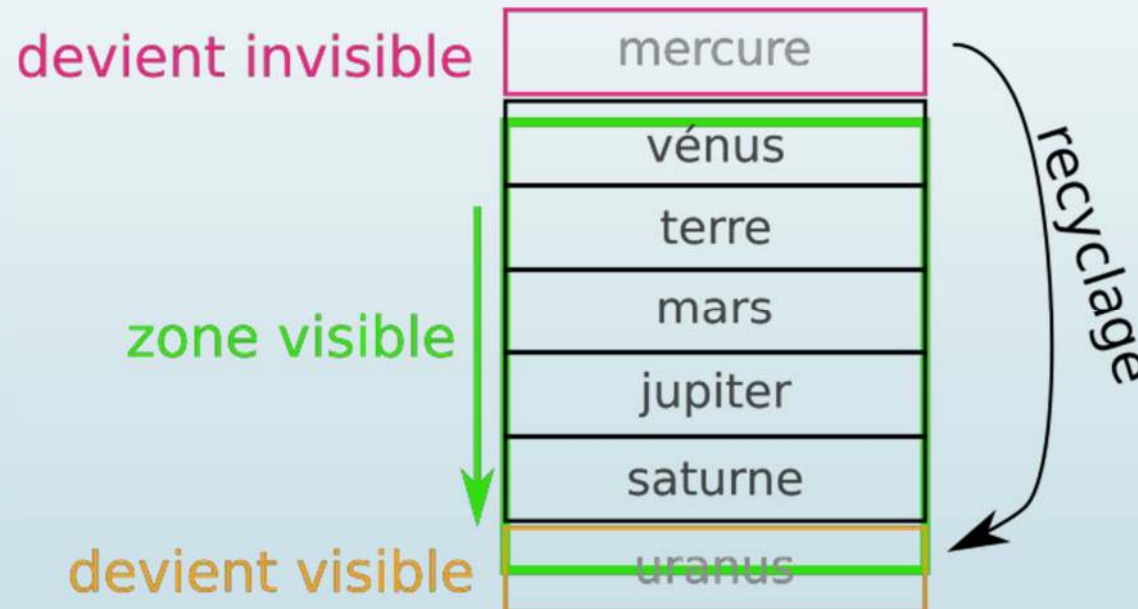
Dans notre activité MainActivity, activité qui contient notre liste

```
public class MainActivity extends AppCompatActivity {  
    ArrayList<ExamClass> list = new ArrayList<>();  
    ....
```

Recyclage des vues

26

Une vue qui devient invisible d'un côté, à cause du scrolling, est renvoyée de l'autre côté, comme sur un tapis roulant, en modifiant seulement son contenu :



Il suffit de remplacer « Mercure » par « Uranus » et de mettre la vue en bas.

Pour permettre ce recyclage, il faut que les vues associées à chaque élément puissent être soit recrées, soit réaffectées. On les appelle des *ViewHolders*, parce que ce sont des mini-containers qui regroupent des vues de base (nom du cours, etc.)

Un *ViewHolder* est instancié pour chaque élément visible de la liste d'items, ex: un exam \longleftrightarrow un *ViewHolder*.

ViewHolder est un objet qui contient le modèle de nos cellules

Le *ViewHolder* est associé à un layout (layout de l'item) géré par un *ViewBinding* pour afficher les informations de l'élément concerné. Pour cela, le *ViewHolder* possède des méthodes pour placer les informations dans ses différentes vues.

Exemple de ViewHolder

28

D'abord, il faut un layout d'item, res/layout/exam_card.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout...>

    <TextView
        android:id="@+id/examName".../>

    <ImageView
        android:id="@+id/examPic".../>

    <TextView
        android:id="@+id/examDate".../>

    <ImageView
        android:id="@+id/examPic2".../>

    <TextView
        android:id="@+id/examMessage".../>

    <TextView
        android:id="@+id/border2".../>

</RelativeLayout>
```

Ce même layout sera instancié pour chaque exam visible dans le RecyclerView. Les TextView seront affectés selon l'examen associée.

On va utiliser son *ViewBinding*, c'est à dire la classe ExamCardBinding, pour accéder facilement aux TextView, ImageView....

Exemple de ViewHolder

29

Le ViewHolder se déclare comme une classe au sein de notre adapter. Voici la classe ExamViewHolder :

```
public class ExamViewHolder
    extends RecyclerView.ViewHolder implements View.OnClickListener {
    private final ExamCardBinding ui;
    public ExamViewHolder(@NonNull ExamCardBinding ui) {
        super(ui.getRoot());
        this.ui = ui;
    }
    public void setExam(ExamClass ex) {
        ui.examName.setText(ex.getName());
        ui.examDate.setText(ex.getDate());
        ui.examMessage.setText(ex.getMessage());
    }
}
```

- La classe ExamViewHolder mémorise un ExamCardBinding, c'est à dire l'ensemble des vues représentant un examen à l'écran, provenant de res/layout/exam_card.xml.
- La méthode setExam met à jour ces vues à partir de la donnée passée en paramètre. Cette méthode est appelée par l'adaptateur lors du recyclage.
- D'autres méthodes seront ajoutées pour gérer les clics sur les éléments.

Rôle d'un adaptateur

L'adaptateur répond à la question que pose la vue : « *que dois-je afficher à tel endroit dans la liste ?* ». Il va chercher les données et instancier ou recycler un *ViewHolder* avec les valeurs.

L'adaptateur est une classe qui :

- stocke et gère les données : liste, connexion à une base de donnée, etc.
- crée et remplit les vues d'affichage des items à la demande du `RecyclerView`.

On retrouve donc ces méthodes dans sa définition

Définition d'un adaptateur

31

Il faut surcharger la classe `RecyclerView.Adapter` qui est une classe générique. Il faut lui indiquer la classe des *ViewHolder*.

Par exemple, `PlaneteAdapter` :

```
public class ExamAdapter extends  
    RecyclerView.Adapter<ExamAdapter.ExamViewHolder> {  
    ... constructeur ...  
    ... surcharge des méthodes nécessaires...
```

Cette classe va gérer l'affichage des éléments individuels et aussi gérer la liste dans son ensemble. Pour cela, on définit un constructeur et on doit surcharger trois méthodes.

Constructeur d'un adaptateur

32

La classe `RecyclerView.Adapter` ne contient aucune structure de donnée. C'est à nous de gérer cela :

```
public class ExamAdapter extends
    RecyclerView.Adapter<ExamAdapter.ExamViewHolder> {

    ArrayList<ExamClass> list = new ArrayList<>();

    public ExamAdapter(ArrayList<ExamClass> list)
    {
        this.list = list;
    }
}
```

La liste est stockée dans l'adaptateur.

NB: c'est un partage de référence, il n'y a qu'une seule allocation en mémoire.

Méthodes à ajouter

33

Pour communiquer avec le RecyclerView, il faut surcharger (redéfinir) trois méthodes.

Pour commencer, celle qui retourne le nombre d'éléments :

```
@Override
public int getItemCount()
{
    return list.size();
}
```

Ensuite, surcharger la méthode qui crée les ViewHolder :

```
@NonNull
@Override
public ExamViewHolder onCreateViewHolder(ViewGroup parent, int viewType)
{
    ExamCardBinding binding =
        ExamCardBinding.inflate(LayoutInflater.from(parent.getContext()),
                                parent, false);
    return new ExamViewHolder(binding);
}
```

Elle est appelée au début de l'affichage de la liste, pour initialiser ce qu'on voit à l'écran.

inflate = transformer un fichier XML en vues Java.

Exécutée lors de la création de notre ViewHolder, cette méthode n'est exécutée qu'une seule fois

Méthodes à ajouter, suite et fin

34

Enfin, surcharger la méthode qui recycle les ViewHolder :

```
@Override
public void onBindViewHolder(final ExamViewHolder viewHolder,
                             final int position)
{
    ExamClass exam = list.get(position);
    viewHolder.setExam(exam);
}
```

- Lors du recyclage d'une cellule, nous recevons comme paramètre sa **position** et **l'instance** du ViewHolder pour mettre à jour la cellule courante
- Cette méthode est appelée pour remplir un ViewHolder avec l'un des éléments de la liste, celui qui est désigné par position

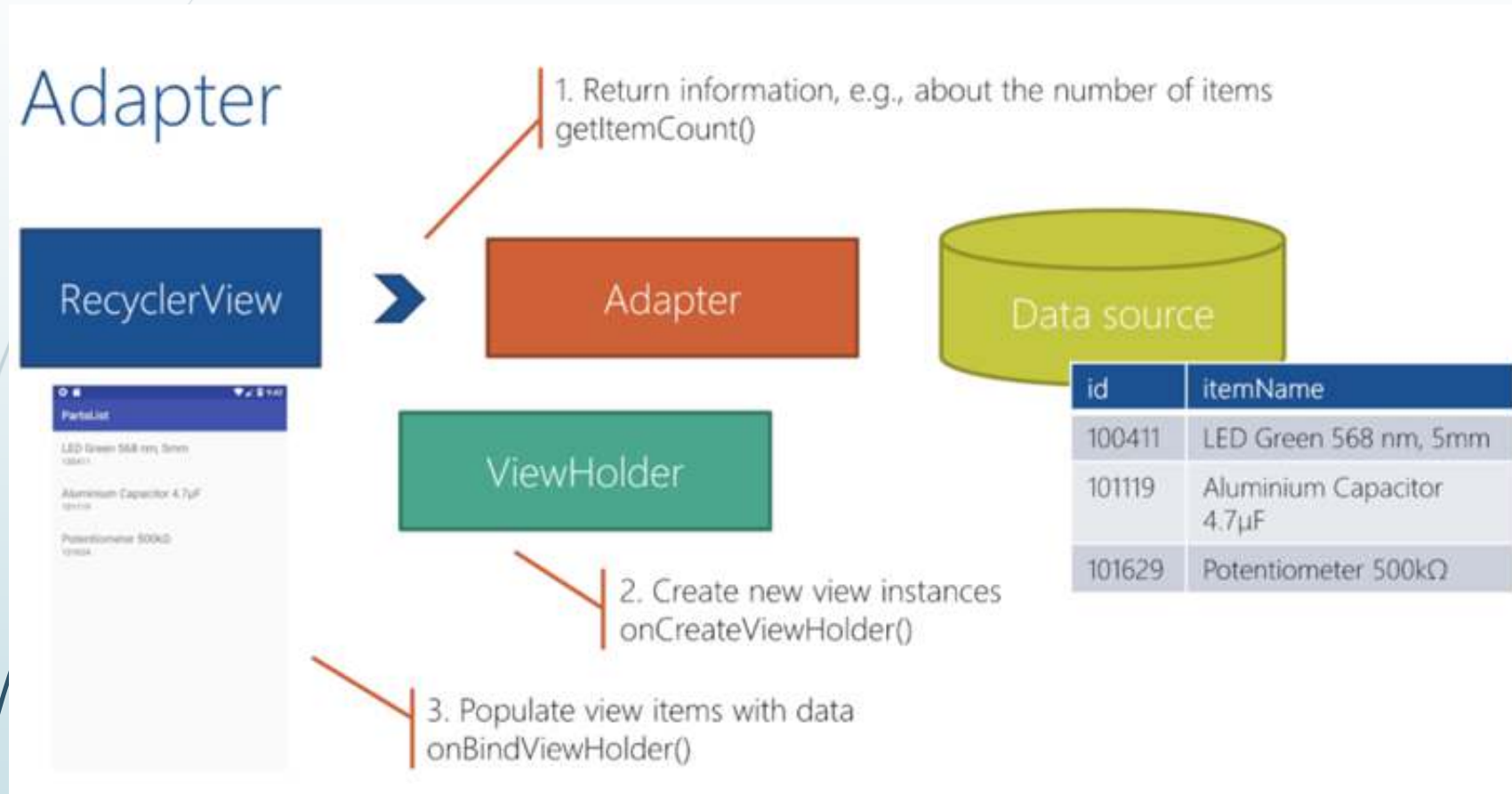
C'est très facile avec le setter (setExam()) définit dans le ViewHolder).

D'autres méthodes seront ajoutées pour gérer les clics sur les éléments.

Communication avec le RECYCLERVIEW

35

Surcharge des méthodes



Configuration de l'affichage

Optimisation du défilement

37

On va s'intéresser à la **mise en page des *ViewHolders*** : en liste, en tableau, en blocs empilés. . . Il suffit seulement de configurer le RecyclerView ; c'est lui qui s'occupe de l'affichage.

D'abord, dans MainActivity, il est important d'indiquer au RecyclerView si les *ViewHolder* ont tous la même taille ou pas :

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    binding = ActivityMainBinding.inflate(getLayoutInflater());
    View view = binding.getRoot();
    setContentView(view);
    list = getData(); //remplir les données de la liste
    adapter= new ExamAdapter(list); //instantier l'adapter avec les éléments de la liste
    binding.recyclerView.setHasFixedSize(true); //recuperer le recyclerview et fixer sa taille
    ....
}
```

Mettre false si les tailles varient d'un élément à l'autre.

LayoutManager

Ensuite, et c'est indispensable, le RecyclerView doit savoir comment organiser les éléments : en liste, en tableau, en grille. . .

Cela se fait avec un *LayoutManager* :

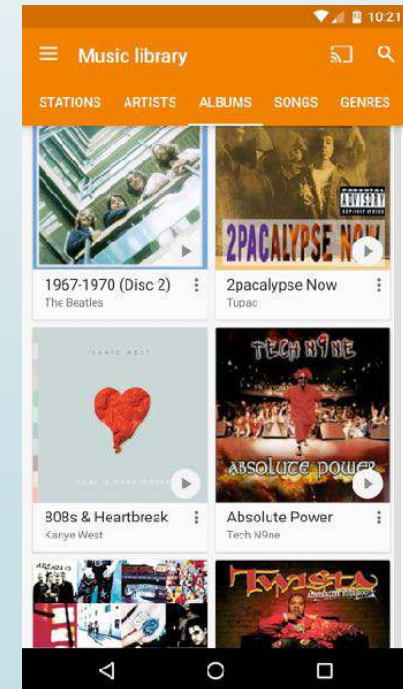
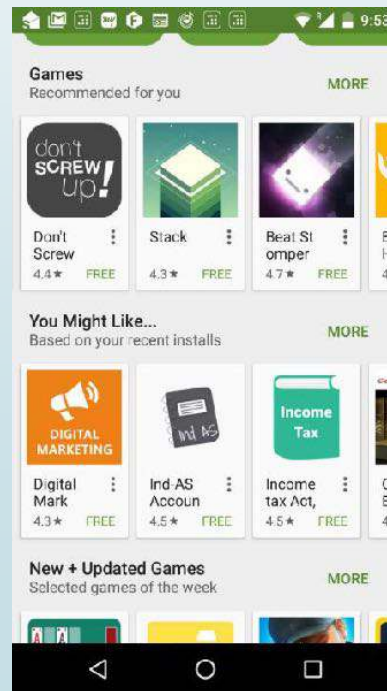
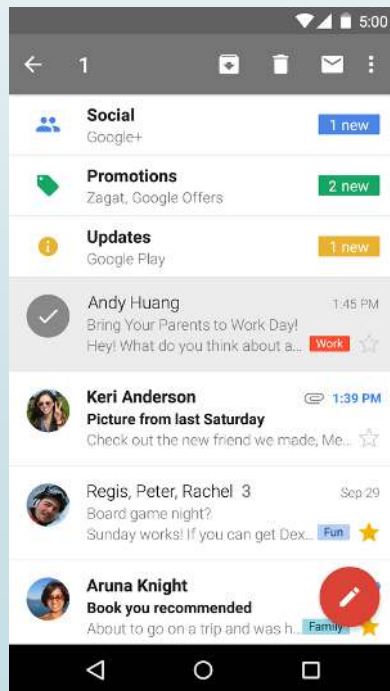
```
// layout manager  
binding.recyclerView.setLayoutManager(  
    new LinearLayoutManager(MainActivity.this));  
binding.recyclerView.setAdapter(adapter); //setAdapter() sur la  
RecyclerView pour lier l'adapter à la liste
```

Sans ces lignes, le RecyclerView n'est pas affiché du tout.

Il existe plusieurs *LayoutManager* qui vont être présentés ci-après.

LayoutManager

- Les LinearLayout sont capable d'afficher les éléments soit de haut en bas (verticalement). Voir l'app Gmail.
- Soit de gauche à droite (horizontalement). Voir l'app Play Store.
- Les GridLayout affichent les éléments sous forme de grille. Voir l'app Play Musique.



LayoutManager

40

Le code utilisé pour un affichage vertical: utiliser la méthode `setLayoutManager()` sur la `RecyclerView` et lui passer en paramètre un nouveau `LinearLayoutManager` qui prends en paramètre le context, l'orientation et un boolean qui indique si vous souhaitez inverser le sens d'affichage de la liste.

▸ Même signature pour un affichage vertical, nous passerons juste notre orientation de `VERTICAL` à `HORIZONTAL`.

```
LinearLayoutManager lm =  
    new LinearLayoutManager(this,  
        RecyclerView.HORIZONTAL, // direction ou VERTICAL  
        false); // sens ou TRUE  
binding.recyclerView.setLayoutManager(lm);
```

Le 3^e paramètre est un booléen qui indique dans quel sens se fait le défilement, vers la droite ou vers la gauche.

Le code utilisé pour un affichage en grille: j'utilise un nouveau `GridLayoutManager` en lui passant comme paramètre le context et le nombre de colonnes.

```
GridLayoutManager gm=new GridLayoutManager(this,2);  
binding.recyclerView.setLayoutManager(gm);
```


Séparateur entre items

41

Par défaut, un RecyclerView n'affiche pas de ligne de séparation entre les éléments. Pour en ajouter une :

```
// séparateur
DividerItemDecoration dividerItemDecoration =
    new DividerItemDecoration(
        this,
        ui.recycler.addItemDecoration(dividerItemDecoration);
```

Actions sur la liste

Avec tout ce qui précède, la liste s'affiche automatiquement et défile à volonté.

On s'intéresse maintenant à ce qui se passe quand on modifie la liste sous-jacente :

- modifications extérieures au RecyclerView, c'est-à-dire le programme Java modifie les données directement dans le `ArrayList`,
- modifications effectuées par le RecyclerView suite aux gestes de l'utilisateur sur les éléments (clics, glissés. . .) voir [ItemAnimator](https://guides.codepath.com/android/using-the-recyclerview#animators)(<https://guides.codepath.com/android/using-the-recyclerview#animators>)

Toute **modification extérieure sur la liste** des éléments doit être signalée à l'adaptateur afin qu'à son tour il puisse prévenir le RecyclerView.

Selon la modification, il faut appeler :

- **notifyItemChanged**(int pos) quand l'élément de cette position a été modifié
- **notifyItemInserted**(int pos) quand un élément a été inséré à cette position
- **notifyItemRemoved**(int pos) quand cet élément a été supprimé
- **notifyDataSetChanged**() si on ne peut pas identifier le changement facilement (tri, réinitialisation. . .)

Défilement vers un élément

45

Pour faire **défiler** afin de **rendre un élément visible**, il suffit d'appeler l'une de ces méthodes sur le RecyclerView :

- **scrollToPosition**(int pos) : fait défiler d'un coup la liste pour que la position soit visible,
- **smoothScrollToPosition**(int pos) : fait défiler la liste avec une animation jusqu'à ce que la position devienne visible.

Clic sur un élément

46

On doit construire soi-même une architecture d'écouteurs. Voici d'abord la situation, pour comprendre la solution.

- 1 L'activité `MainActivity` veut être prévenue quand l'utilisateur clique sur un élément de la liste.
- 2 L'objet qui reçoit les événements utilisateur est le *ViewHolder*. Il suffit de lui ajouter la méthode `onClick(View v)` de l'interface `View.OnClickListener` (comme un simple `Button`) pour être prévenu d'un clic.
- 3 Un `RecyclerView` regroupe plusieurs *ViewHolder* ; chacun peut être cliqué (un à la fois). Celui qui est cliqué peut faire quelque chose dans sa méthode `onClick`, mais le problème, c'est que le *ViewHolder* ne connaît pas l'activité à prévenir.

Il faut donc faire le lien entre ces *ViewHolder* et l'activité. Ça va passer par l'adaptateur, le seul qui soit au contact des deux.

- 1 Il faut que l'activité définisse un écouteur de clics et le fournisse à l'adaptateur. Tant qu'à faire, on peut définir notre propre sorte d'écouteur qui recevra la position de l'objet cliqué en paramètre (c'est le plus simple à faire).
- 2 L'adaptateur transmet cet écouteur à tous les *ViewHolder* qu'il crée ou recycle.
- 3 Chaque *ViewHolder* possède donc cet écouteur et peut le déclencher 4 le cas échéant.
- 5 L'activité, réveillée, accède à la donnée concernée (position)

Voyons comment créer notre propre type d'écouteur, puis quoi en faire.

Notre écouteur de clics

48

Il suffit d'ajouter une interface publique dans l'adaptateur :

```
public class ExamAdapter extends  
RecyclerView.Adapter<ExamAdapter.ExamViewHolder> {  
  
    private OnItemClickListener listener;  
  
    public interface OnItemClickListener {  
        void onItemClick(int position);  
    }  
  
    public void setOnItemClickListener(OnItemClickListener onItemClickListener) {  
        this.listener = onItemClickListener;  
    }  
}
```


Notre écouteur de clics

49

Il faut maintenant que l'adaptateur fournisse cet écouteur aux *ViewHolders* :

```
@Override
public void onBindViewHolder(final ExamViewHolder viewHolder,
                             final int position)
{
    .....
    viewHolder.setOnItemClickListener((OnItemClickListener)
    this.listener);
}
```

Dans le *ViewHolder*, il y a le même *setter* et la même variable. L'adaptateur se contente de fournir l'écouteur à chacun.

Notre écouteur de clics, suite

50

Les *ViewHolders* doivent mémoriser cet écouteur :

```
public class ExamViewHolder
    extends RecyclerView.ViewHolder implements View.OnClickListener {
    private final ExamCardBinding ui;
    private ExamAdapter.OnItemClickListener listener;
    ....
    public void setOnItemClickListener(OnItemClickListener l) {
        this.listener = l;
    }
    .....
```

Remarquez comment on fait référence à l'interface définie dans la classe `ExamAdapter`.

Les *ViewHolders* doivent aussi recevoir les événements puis déclencher l'écouteur :

```
public ExamViewHolder(@NonNull ExamCardBinding ui) {  
    super(ui.getRoot());  
    this.ui = ui;  
    itemView.setOnClickListener(this);  
}  
@Override  
public void onClick(View v) {  
    if(listener != null)  
        listener.onItemClick(getAdapterPosition());  
}
```

4

La méthode `getAdapterPosition()` retourne la position de ce *ViewHolder* dans son adaptateur.

Notre écouteur de clics, suite et fin

52

L'adaptateur définit une interface que d'autres classes vont implémenter, par exemple une référence de méthode de l'activité :

```
public class MainActivity extends AppCompatActivity {
    ....
    ExamAdapter adapter;
    @Override
    protected void onCreate(Bundle savedInstanceState)
    {
        ...
        //adapteur
        adapter= new ExamAdapter(list);
        binding.recyclerView.setHasFixedSize(true);

        binding.recyclerView.setLayoutManager(lm);
        binding.recyclerView.setAdapter(adapter);

        //click RV: écouteur pour les clics sur les éléments de la liste
        adapter.setOnItemClickListener(new ExamAdapter.OnItemClickListener() {
            @Override
            public void onItemClick(int position) {
                ExamClass model = list.get(position); //L'activité, réveillée, accède à la donnée concernée
                //.....UTILISER LE MODEL:EXAM CLASS ELEMENT.....
            }
        });
    }
}
```

Schéma récapitulatif

53

Ça commence à gauche dans l'activité. Son écouteur est transmis via l'adaptateur à tous les *ViewHolders*. L'un d'eux est activé par un clic et déclenche l'écouteur. L'activité, réveillée, accède à la donnée concernée.

