



Institut
Supérieur de
Technologies

BURKINA FASO

UNITÉ-PROGRÈS-JUSTICE

MASTER2: RÉSEAUX INFORMATIQUES ET MULTIMÉDIA

PROGRAMMATION MOBILE

Les Vues à adaptateur et les boîtes de dialogue

COMPAORE MOCTAR

4 May 2022

PLAN

- ✓ Vues à adaptateur (AdapterView)
- ✓ Listeners d'un AdapterView
- ✓ Vues à adaptateur personnalisé
- ✓ Boites de dialogue (Dialog)
- ✓ AlertDialog
- ✓ ProgressDialog
- ✓ Dialog personnalisés

VUES À ADAPTATEUR (ADAPTERVIEW)

Une vue à adaptateur (**AdapterView**) est une vue complexe (sous-classe de ViewGroup) qui contient plusieurs vues, utilisée souvent pour afficher des collections de données (List, Set, Map, ...).

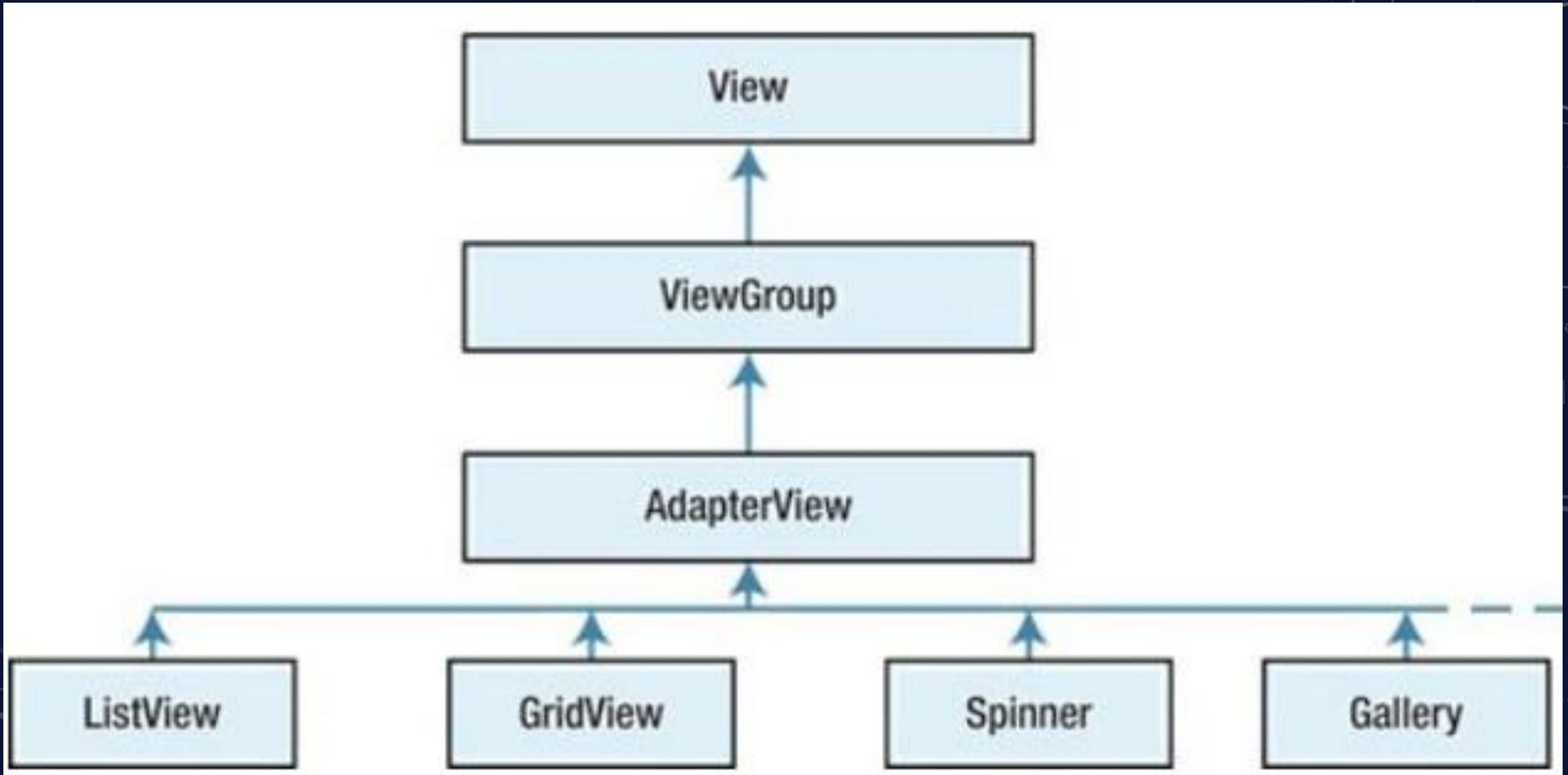
Les vues filles sont déterminées par un adaptateur (**Adapter**) qui relie la vue à adaptateur aux données.

VUES À ADAPTATEUR (ADAPTERVIEW)

Les vues à adaptateur les plus connues sont :

- ✓ **ListView** : affiche une liste d'éléments avec un défilement vertical,
- ✓ **GridView** : affiche des données sur une grille avec défilement vertical où les attributs `columnWidth` et `numColumns` permettent de configurer l'alignement des cellules,
- ✓ **Spinner** : est une liste déroulante de données à choix unique,
- ✓ **Gallery** : affiche une liste éléments avec un défilement horizontal qui verrouille l'élément sélectionné au centre ,
- ✓ **AutoCompleteTextView** : permet d'obtenir des suggestions, lorsque on écrit du texte,
- ✓ **RecyclerView** : permet d'afficher un grand nombre de données en améliorant les performances

VUES À ADAPTATEUR (ADAPTERVIEW)

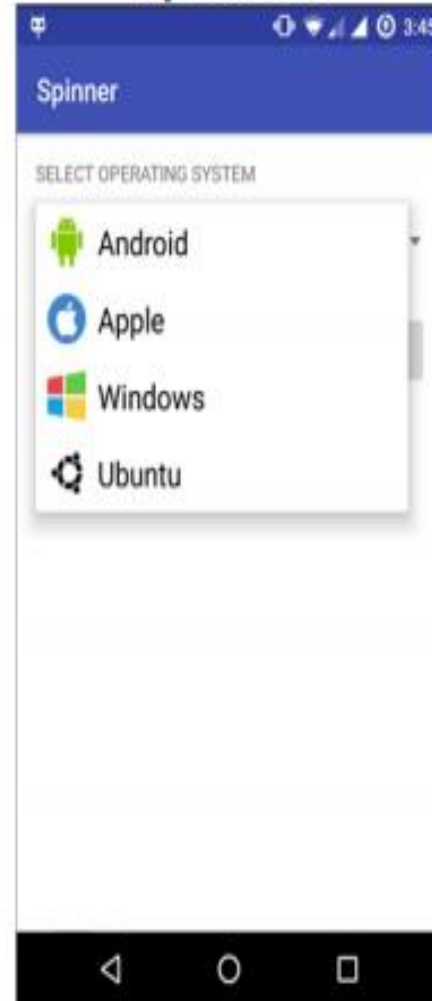


VUES À ADAPTATEUR (ADAPTERVIEW)

ListView



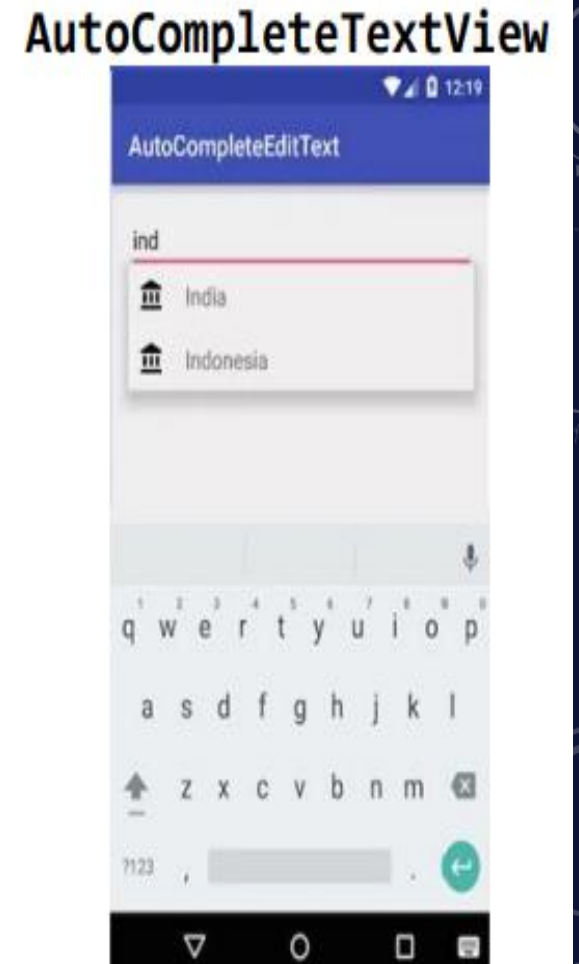
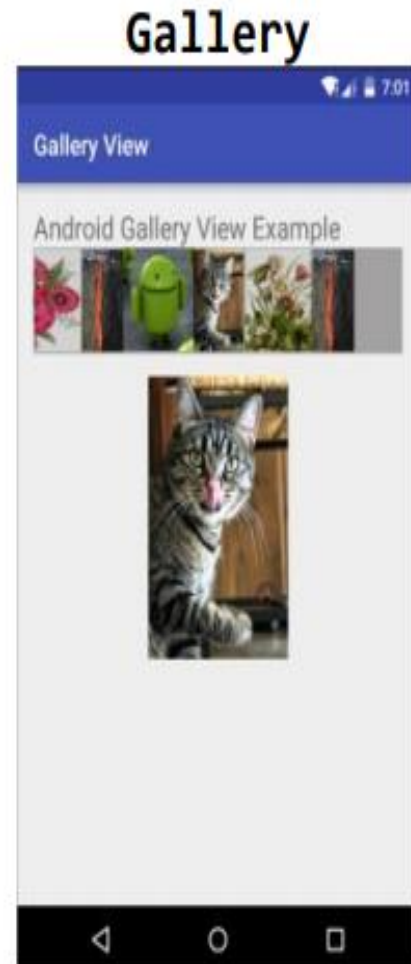
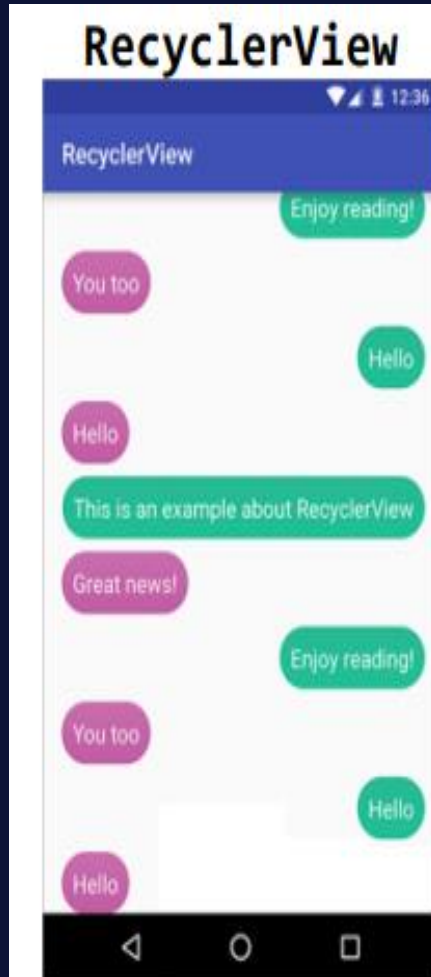
Spinner



GridView



VUES À ADAPTEUR (ADAPTERVIEW)

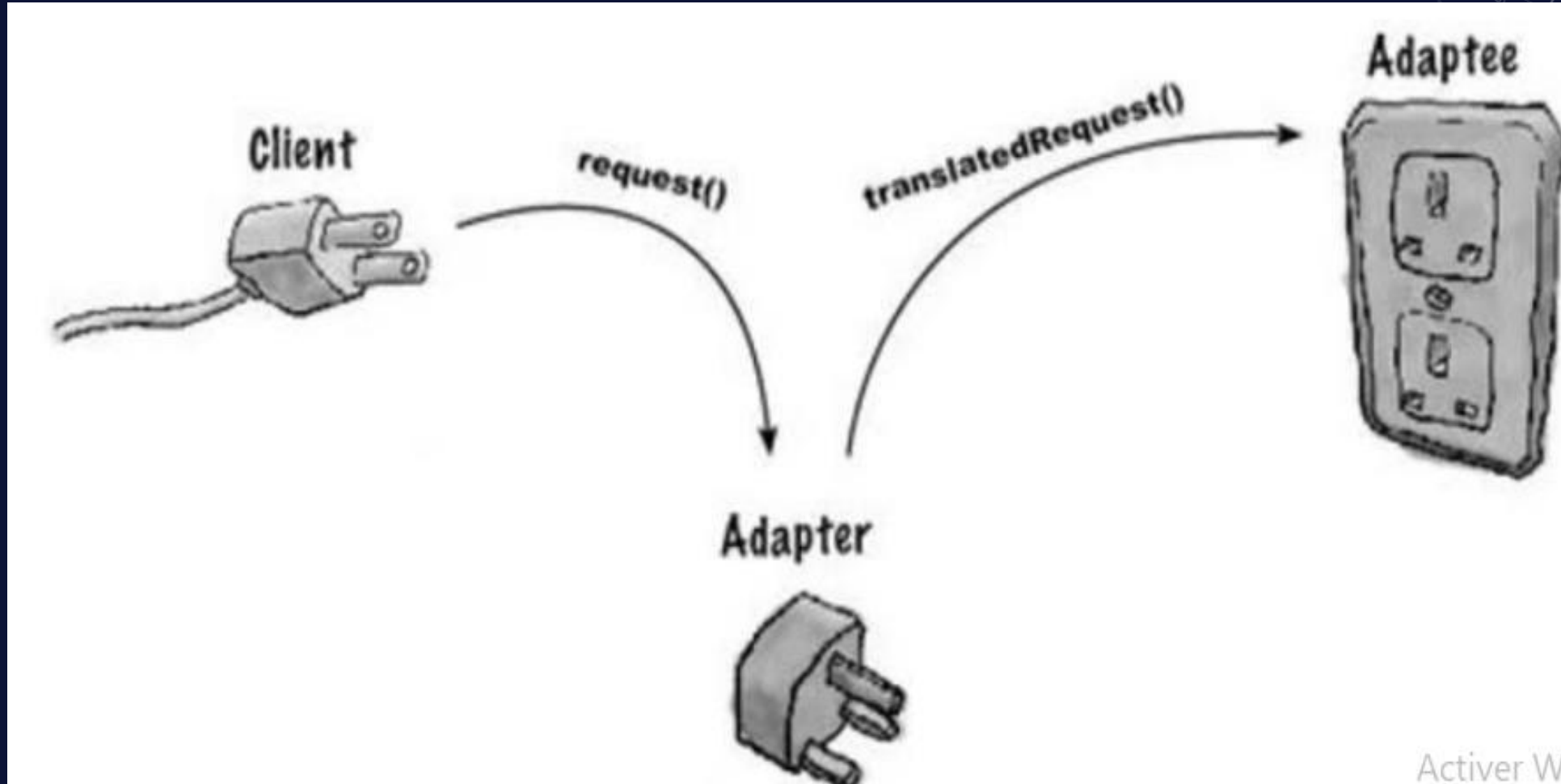


DESIGN PATTERN : ADAPTER

La vue qui permet d'afficher une liste se basent sur le design pattern Adapter pour remplir la vue. Ce pattern est utilisé dans toute l'architecture Android pour remplir des listes, de ce fait, il est indispensable d'en connaître son fonctionnement.

Le client créer une demande en appelant une méthode **request()** d'un adaptateur. Ce dernier traduit cette demande en un ou plusieurs appels sur l'adaptée en utilisant une méthode **translateRequest()**. Le client reçoit les résultats de l'appel et ne sait jamais qu'il y a un adaptateur qui effectue la traduction.

DESIGN PATTERN : ADAPTER

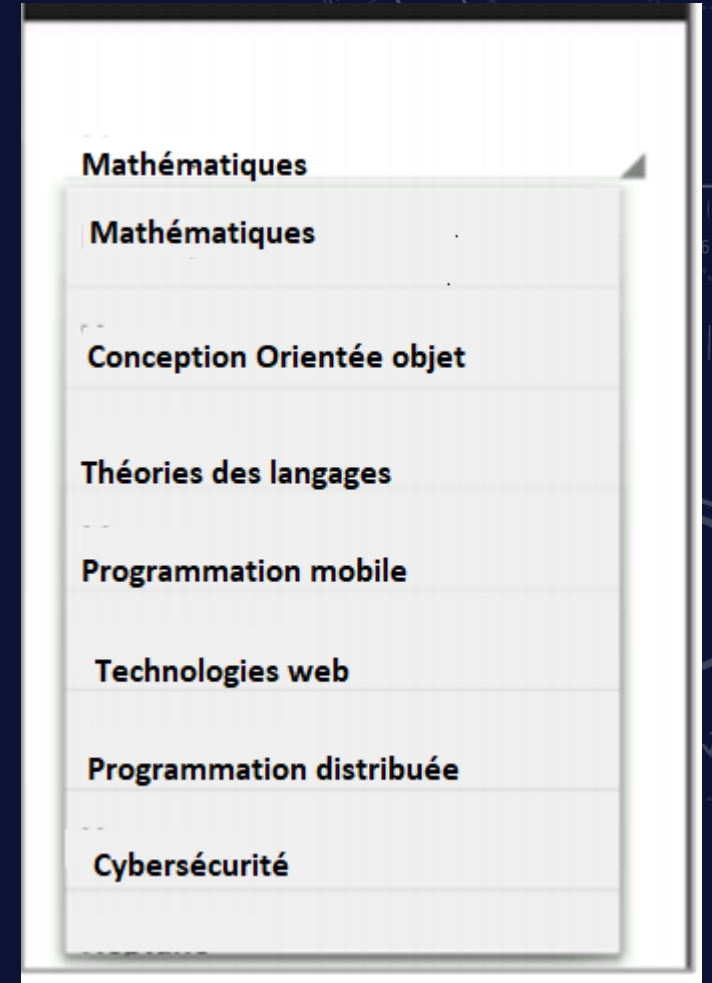


Donc dans notre contexte, un adaptateur permet de transformer une collection de données (un tableau, une liste ou un curseur de données) en widgets pour les insérer dans une vue à adapter (**android.widget.AdapterView**).

DESIGN PATTERN : ADAPTER

On propose de créer une liste déroulante semblable à la figure suivante. Pour ce faire, il faut rajouter un widget de type Spinner dans le layout. Ensuite, on remplit le spinner avec les données en utilisant un adaptateur (**android.widget.Adapter**)

```
<Spinner  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/spinner" />
```



DESIGN PATTERN : ADAPTER

- L'adaptateur le plus utilisé sous Android est `ArrayAdapter<T>` qui est dédié aux tableaux.
- Par défaut, Android fournit un layout prédéfini qui contient un `TextView` (`android.R.layout.simple_list_item_1`), permettant à l'adaptateur de créer une vue pour chaque élément du tableau en appelant la méthode `toString()` de chaque élément, et en plaçant son contenu dans un objet `TextView`.
- Dans le code suivant, un `ArrayAdapter<String>` est créé à l'aide de son constructeur qui comprend 3 paramètres : le contexte de l'application (généralement l'activité dans laquelle est défini l'`AdapterView`, le layout de chaque élément et le tableau contenant les données.
- Enfin, l'adaptateur créé est associé au Spinner, en utilisant la méthode `setAdapter(...)`.

DESIGN PATTERN : ADAPTER

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);

String[] items = {"-Choisir-", "Mathématiques", "Programmation  
Mobile", "Conception orientée objet", "Technologie  
web", « Cybersécurité" };

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

Pratiques

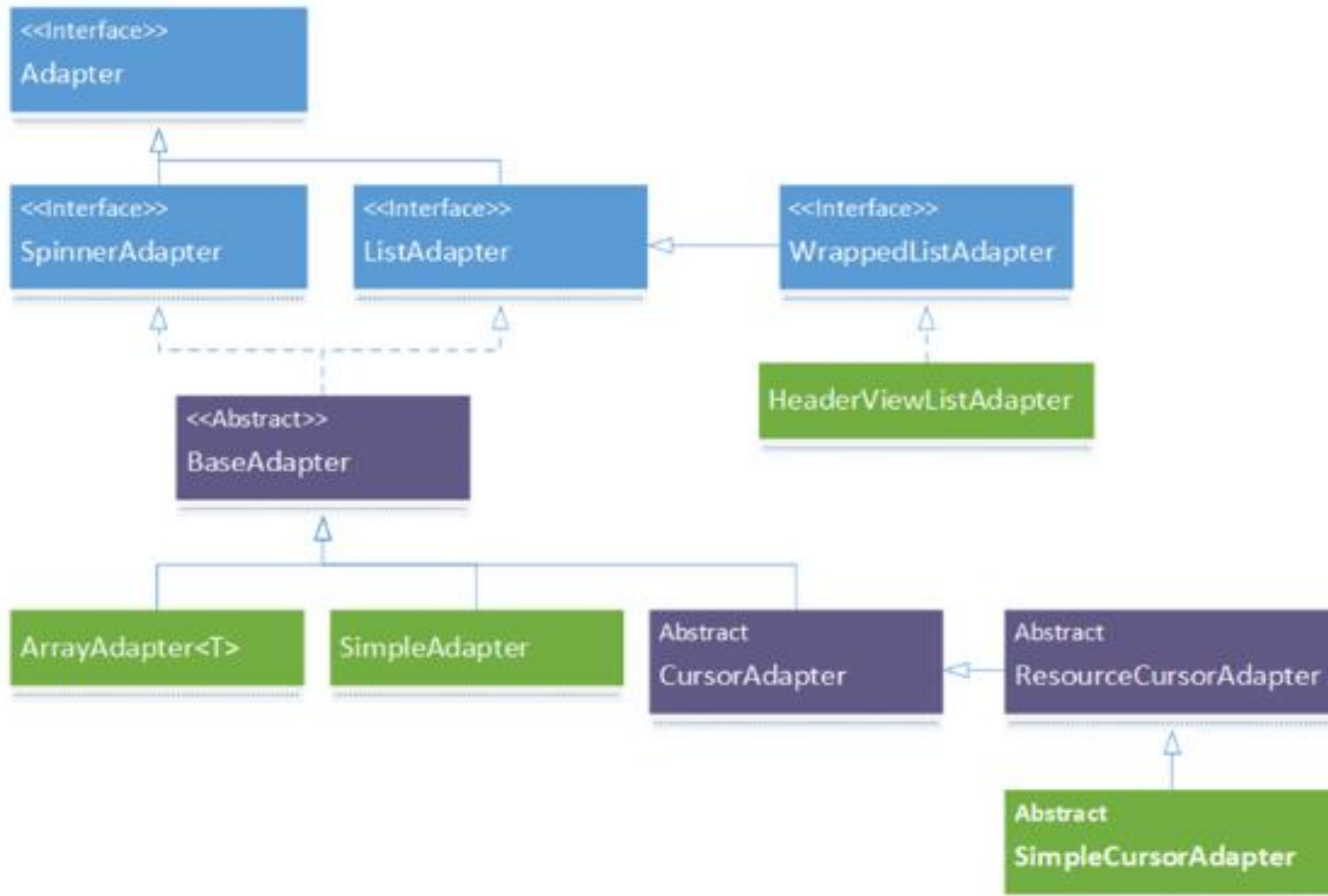
The background is a dark blue gradient. It features several abstract geometric elements: a large circle with radial lines and degree markings (0 to 210) in the top right; a smaller circle with radial lines in the bottom right; and a partial circle with radial lines in the bottom left. There are also some faint, larger concentric circles and radial lines in the top left.

Hiérarchie des adaptateurs

Android fournit différents adaptateurs qui permettent de représenter les informations de façon standard dans une vue à adaptateur :

- ✓ **ArrayAdapter<T>** : pour tous les types de tableaux et de listes,
- ✓ **BaseAdapter** : sert à implémenter des adaptateurs personnalisés,
- ✓ **CursorAdapter** : pour traiter les données de type Cursor,
- ✓ **HeaderViewListAdapter** : permet d'ajouter des entêtes et pieds de page aux ListView,
- ✓ **ResourceCursorAdapter** : sert à créer des vues à partir d'une disposition XML,
- ✓ **SimpleAdapter** : pour afficher des données complexes (par exemple un tableau de tableaux),
- ✓ **SimpleCursorAdapter** : pour adapter les données d'un Cursor de Base de données.

Hiérarchie des adaptateurs



Hiérarchie des adaptateurs

Quelques méthodes communes à tous les adaptateurs permettant de mettre à jour les données.

```
void add(T item) // ajouter un élément en fin de l'AdapterView
void insert(T item, int index) // insérer un élément à une position donnée
void addAll(T... items) // insérer plusieurs éléments
T getItem(int index) // récupérer l'élément d'une position donnée
int getPosition(Object o) // récupérer la position d'un élément donné
void remove(T item) // supprimer un élément donnée
void clear() // supprimer tous les éléments
void notifyDataSetChanged() // notifie l'AdapterView des nouveaux
changements pour se ra
```

Listeners d'un AdapterView

Un **AdapterView** est conçu pour afficher une liste d'éléments à l'utilisateur et les actions que peut effectuer ce dernier est le clic ou la sélection d'un élément de **l'AdapterView**. Afin d'interagir avec un **AdapterView**, il suffit d'intercepter l'événement déclenché (par exemple, un clic sur un élément), à l'aide de listeners (écouteurs). Voici les listeners proposés pour un

AdapterView :

- ✓ OnItemClickListener : pour intercepter l'évènement du clic sur un élément. La méthode à surcharger est :
 - ✓ onItemClick(AdapterView<?> parent, View view, int position, long id)

Listeners d'un AdapterView

- ✓ **OnItemClickListener** : pour intercepter l'évènement du clic sur un élément. La méthode à surcharger est : **onItemClick**(AdapterView<?> parent, View view, int position, long id)
- ✓ **OnItemLongClickListener** : pour intercepter l'évènement du clic long sur un élément. La méthode à surcharger est : **onLongClick**(AdapterView<?> parent, View view, int position, long id)
- ✓ **OnItemSelectedListener** : pour intercepter l'évènement de la sélection d'éléments. Les méthodes à surcharger sont :
onItemSelected(AdapterView<?> parent, View view, int position, long id)
onNothingSelected(AdapterView<?> parent)

Listeners d'un AdapterView

Pour ajouter un **listener** XXX à un **AdapterView**, il suffit d'utiliser la méthode `setXXX(...)`.

Par exemple, pour ajouter un comportement au clic sur un élément de **l'AdpaterView**, il faut définir un **OnItemClickListener** et surcharger la méthode **onItemClick(...)**.

L'élément cliqué et sa position sont capturés par les paramètres `view` et `position`

Listeners d'un AdapterView

```
spinner.setOnItemClickListener(new AdapterView.OnItemClickListener(){  
  
    @Override  
    public void onItemClick(AdapterView<?> parent, View view, int position, long  
id){  
        String planetteClique= items[position];  
        Toast t= Toast.makeText(getApplicationContext(), "Le Module cliqué est  
".concat(planetteClique), Toast.LENGTH_LONG);  
        t.show();  
    }  
});
```

Pratiques

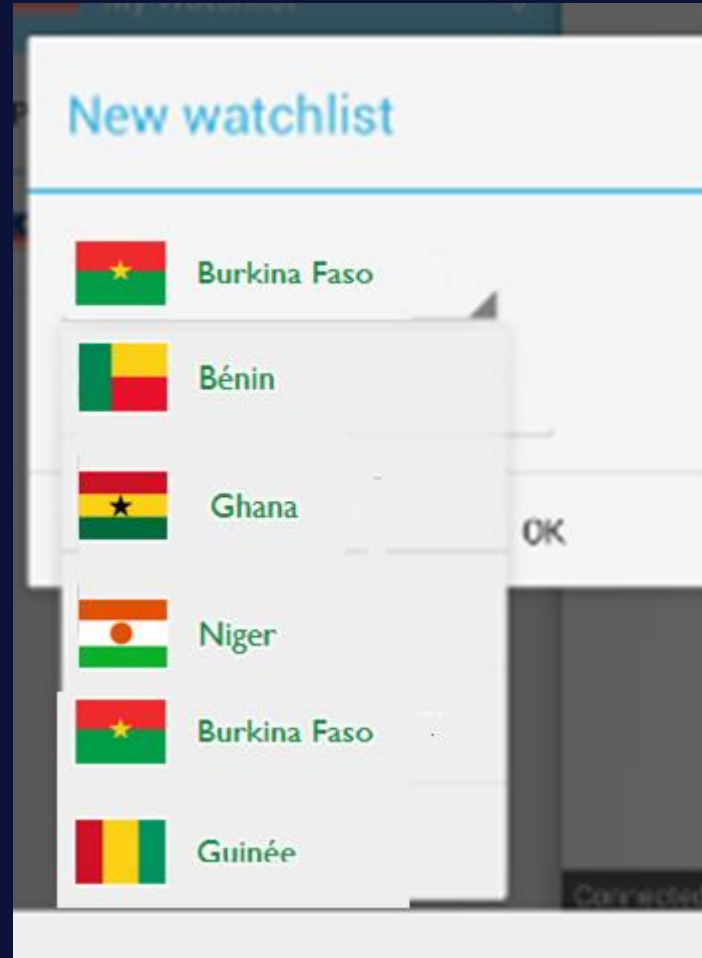
The background is a dark blue gradient. It features several decorative elements: a large circular scale on the right side with markings from 0 to 210 degrees; a smaller circular scale in the top left corner; and two circular patterns in the bottom left corner, one with a dashed arrow pointing clockwise and another with a solid arrow pointing counter-clockwise.

Vues à adaptateur personnalisé

- ✓ Dans certaines situations, les adaptateurs fournis par Android ne permettent pas de résoudre un cas particulier.
- ✓ Alors, il est possible de représenter autre chose qu'un `TextView` dans un élément d'un `AdapterView`.
- ✓ Pour ce faire, il suffit de créer un adaptateur, dit personnalisé, soit en le dérivant d'une classe concrète existante, par exemple `ArrayAdapter<T>`, ou soit en dérivant d'une classe abstraite, par exemple `BaseAdapter`, et ainsi redéfinir la méthode `getView(...)` qui permet de retourner à l'`AdapterView` chaque élément à afficher.

Vues à adaptateur personnalisé

- ✓ Dans l'exemple suivant, nous allons élaborer un Spinner à adaptateur personnalisé affichant une liste de pays avec leurs drapeaux. Voici les étapes à suivre :



Vues à adaptateur personnalisé

✓ Création de l'entité

Afin de stocker et de manipuler les informations d'un pays, on crée la classe Country contenant le nom du pays (name), l'identifiant concernant l'image de son drapeau (flagResourceId) et un constructeur pour instancier cette classe

```
public class Country {  
    private int flagResourceId;  
    private String name;  
  
    public Country(int flagResourceId, String name){  
        this.flagResourceId = flagResourceId;  
        this.name = name;  
    }  
}
```

Vues à adaptateur personnalisé

- ✓ Elaboration d'une vue pour chaque élément

Il est indispensable d'élaborer la vue de chaque élément du Spinner pour organiser les informations de chaque pays à afficher. Pour ce faire, on crée un layout appelé `item_country`, qui comporte un `ImageView` et un `TextView` pour afficher respectivement le drapeau et le nom du pays.

Vues à adaptateur personnalisé

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content">
    <ImageView
        android:id="@+id/flagIV"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
    <TextView
        android:id="@+id/nameTV"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content" />
</LinearLayout>
```

Vues à adaptateur personnalisé

- ✓ Création d'un adaptateur personnalisé

Dans notre exemple, on choisit de créer un adaptateur personnalisé, dérivant de la classe `ArrayAdapter<T>`, ce qui impose l'implémentation d'un constructeur et la redéfinition de la méthode `getView(int position, View convertView, ViewGroup parent)`.

Vues à adaptateur personnalisé

```
public class CountryAdapter extends ArrayAdapter<Country> {
    Activity activity;
    int itemResourceId;
    List<Country> items;

    public CountryAdapter(Activity activity, int itemResourceId, List<Country> items){
        super(activity, itemResourceId, items);
        this.activity = activity;
        this.itemResourceId = itemResourceId;
        this.items = items;
    }

    @Override
    public View getView(int position, View convertView, ViewGroup parent) {
        View layout = convertView;
        if(convertView == null){
            LayoutInflater inflater = activity.getLayoutInflater();
            layout = inflater.inflate(itemResourceId, parent, false);
        }
        TextView nameTV = (TextView) layout.findViewById(R.id.nameTV);
        ImageView flagIV = (ImageView) layout.findViewById(R.id.flagIV);
        nameTV.setText(items.get(position).name);
        flagIV.setImageResource(items.get(position).flagResourceId);

        return layout;
    }
}
```

Vues à adaptateur personnalisé

Implémenter le Constructeur

Le constructeur de l'adaptateur personnalisé doit avoir au moins 3 paramètres (le **context**, le **layout** et les **données**) afin de les relayer à son parent (**ArrayAdapter<T>**) en utilisant la méthode **super(...)**. Par ailleurs, ces paramètres sont stocké dans des variables d'instance.

Vues à adaptateur personnalisé

Redéfinir la méthode `getView(...)`

La méthode `getView(...)` permet de retourner un objet `View`, pour chaque élément de `l'AdapterView`. Cette méthode est appelée à chaque fois qu'un élément de `l'AdapterView` est affiché à l'écran

Vues à adaptateur personnalisé

Redéfinir la méthode `getView(...)`

- ✓ Obtenir le layout : Le paramètre `convertView` de la méthode `getView(...)` est la vue qui est employée pour représenter les informations de l'élément d'indice `position`.
- ✓ Pour économiser les ressources, Android gère un cache de ces vues : Si `convertView` est égal à `null` (c'est-à-dire la vue n'existe pas encore dans le cache), alors il faut créer une nouvelle vue à l'aide d'un `LayoutInflater`.
- ✓ Un `LayoutInflater` permet de convertir les éléments d'un fichier layout XML en un arbre d'objets de type `View`.

Vues à adaptateur personnalisé

Redéfinir la méthode getView(...)

- ✓ Récupérer les vues : Le paramètre **convertView** est bien réutilisé lorsqu'il existe, cependant, lors de chaque appel à **getView(...)**, les références du **TextView** et de **l'ImageView** sont recalculées.
- ✓ Remplir les vues : Enfin, le **TextView** et **l'ImageView** sont instanciés par les informations de l'élément d'indice position.

Vues à adaptateur personnalisé

Associer l'adaptateur au Spinner

```
<Spinner  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/spinner" />
```

Vues à adaptateur personnalisé

Associer l'adaptateur au Spinner

Enfin, l'adaptateur personnalisé est instancié par le contexte, le layout de chaque élément ainsi que les données qui sont stockés dans une `List<Country>`. Ensuite, il est associé au Spinner, en utilisant la méthode `setAdapter(...)`.

```
<Spinner  
    android:layout_width="match_parent"  
    android:layout_height="wrap_content"  
    android:id="@+id/spinner"/>
```

Vues à adaptateur personnalisé

Associer l'adaptateur au Spinner

```
Spinner spinner = (Spinner) findViewById(R.id.spinner);  
...  
List<Country> items = new ArrayList<>();  
items.add(new Country(R.drawable.flag_faso, "Burkina Faso"));  
...  
CountryAdapter adapter = new CountryAdapter( MainActivity.this,  
R.layout.item_country, items);  
spinner.setAdapter(adapter);
```

Boîtes de dialogues

Une boîte de dialogue est une petite fenêtre qui passe au premier plan pour informer l'utilisateur ou interagir avec lui.

Cette fenêtre est modale car l'utilisateur ne peut faire aucune action en dehors de celle-ci.

Sous Android, toutes les boîtes de dialogues dérivent de la classe `android.app.Dialog` et sont toujours créées à l'intérieur d'une activité.

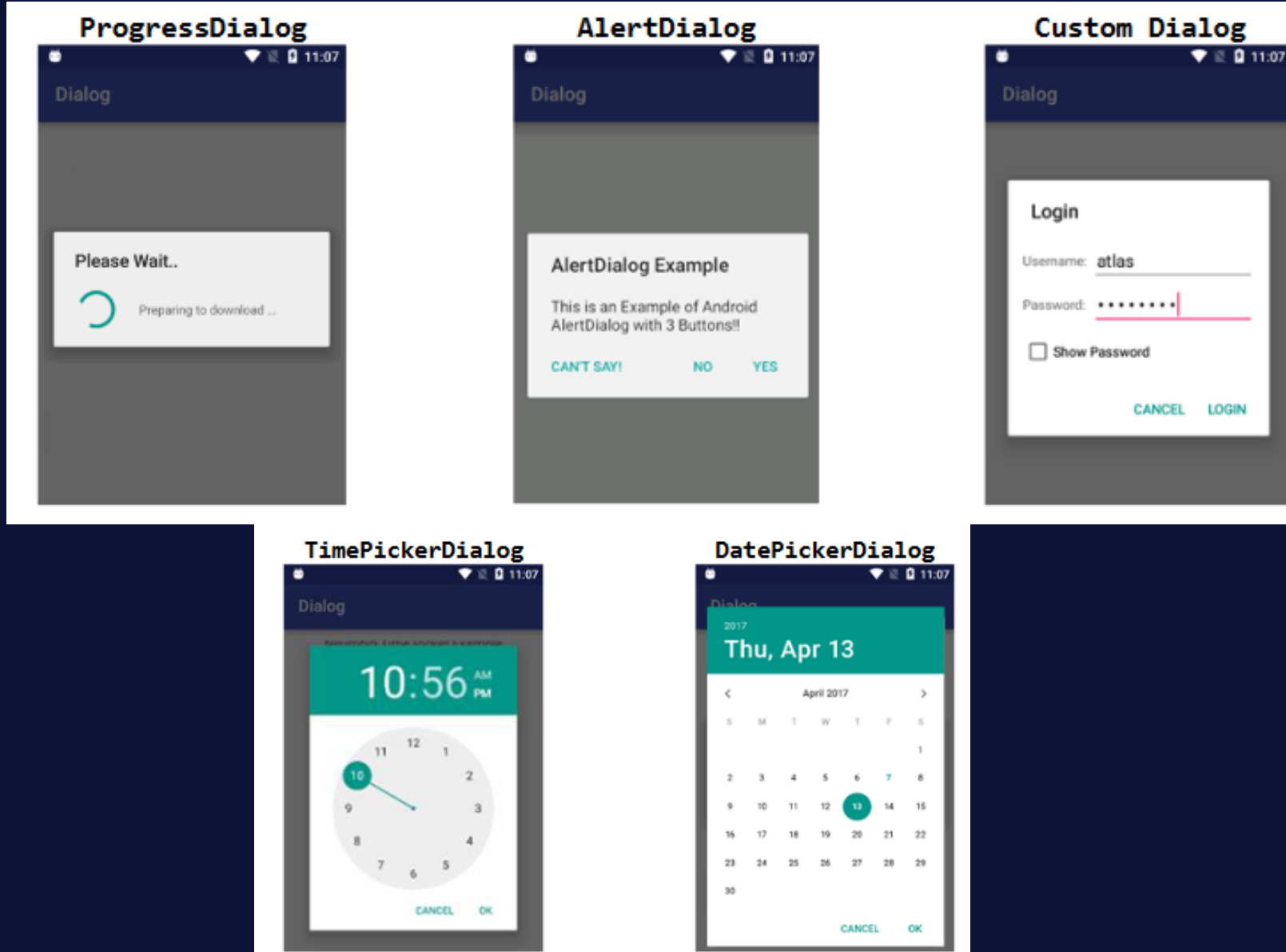
Boîtes de dialogues

Android fournit les boîtes de dialogues prédéfinies suivantes :

- ✓ **AlertDialog** : permet d'afficher des informations d'alerte ou obtenir une confirmation de l'utilisateur.
- ✓ **ProgressDialog** : consiste à faire patienter l'utilisateur ou lui afficher l'avancement d'un traitement en cours,
- ✓ **DatePickerDialog** : permet à l'utilisateur de définir une date donnée,
- ✓ **TimePickerDialog** : permet à l'utilisateur de définir une heure donnée.

Boîtes de dialogues

Android fournit les boîtes de dialogues prédéfinies suivantes :



Boîtes de dialogues

AlertDialog

Un **AlertDialog** est une boîte de dialogue qui permet d'afficher un titre, un texte, une liste d'éléments et des boutons.

Donc, pour construire une AlertDialog, Android fournit la classe **AlertDialog.Builder**, qui permet de simplifier énormément la construction à travers plusieurs méthodes, telles que **setTitle(...)** et **setMessage(...)** qui permettent respectivement de définir le titre et le message de la boîte de dialogue.

Boîtes de dialogues

AlertDialog

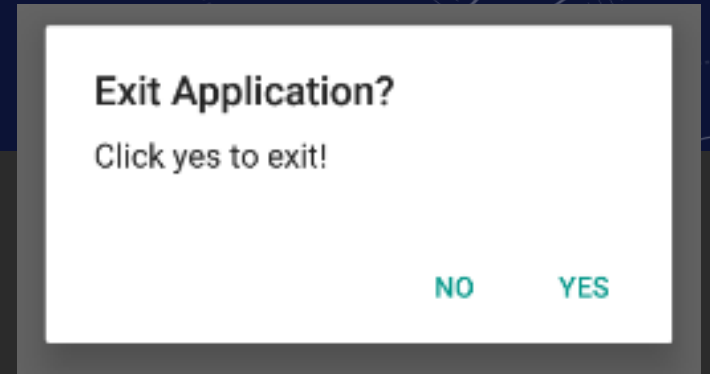
Dans un AlertDialog, il est possible d'ajouter jusqu'à 3 boutons pour interagir avec l'utilisateur en utilisant les méthodes `setPositiveButton(...)`, `setNeutralButton(...)` et `setNegativeButton(...)`.

Le traitement de la réponse est fait de manière asynchrone, ce qui impose l'utilisation des callback. Dans l'exemple suivant, on crée une boîte de dialogue de type **Yes/No**, donc il faut définir 2 listeners pour traiter chacune des réponses **Yes/No**.

Boîtes de dialogues

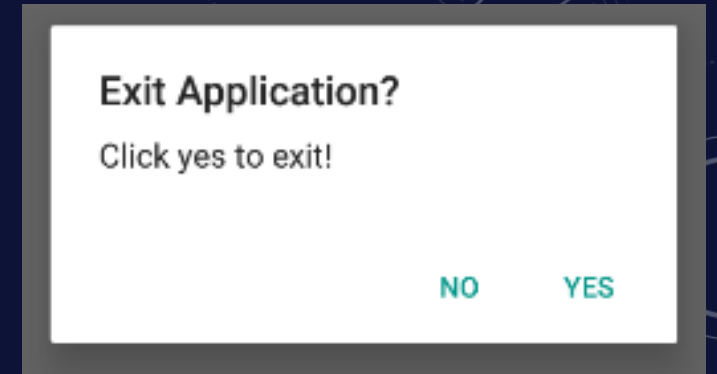
AlertDialog

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Exit Application?");
builder.setMessage("Click yes to exit!");
builder.setPositiveButton("Yes", new DialogInterface.OnClickListener(){
    @Override
    public void onClick(DialogInterface dialog, int id) {
        ...
    }
});
builder.setNegativeButton("No", new DialogInterface.OnClickListener(){
    @Override
    public void onClick(DialogInterface dialog, int id) {
        ...
    }
});
AlertDialog dialog = builder.create();
dialog.show();
...
dialog.cancel();
```



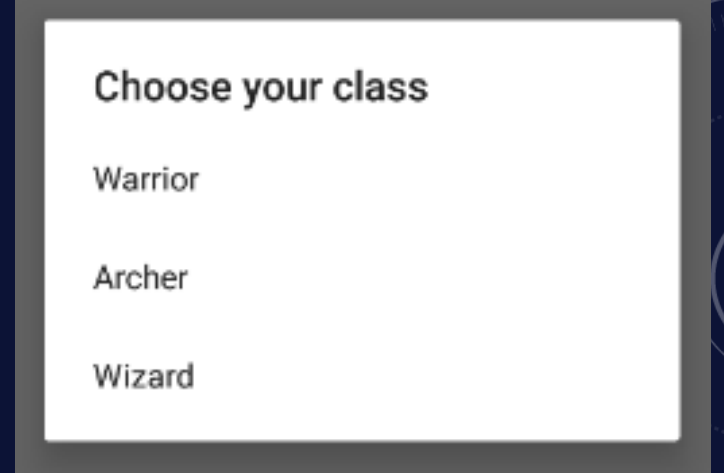
Boites de dialogues

AlertDialog



Après la création de la boite de dialogue, l'affichage de la boite de dialogue est effectué avec la méthode **show()** et sa fermeture est réalisée en utilisant la méthode **dismiss()** ou **cancel()**.

Boites de dialogues



AlertDialog avec choix

Il est possible de mettre une liste d'éléments dans un AlertDialog en utilisant la méthode **setItems(...)**. Le 2ème paramètre de cette méthode permet de définir le listener qui traite le clic sur chaque élément de la list,

Boites de dialogues

AlertDialog avec choix

```
String[] list = {"Warrior", "Archer", "Wizard"};
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Choose your class");
builder.setItems(list, new DialogInterface.OnClickListener() {
    @Override
    public void onClick(DialogInterface dialog, int position) {
        Log.i("Dialog", "Position = " + position);
    }
});
AlertDialog dialog = builder.create();
dialog.show();
```

Choose your class

Warrior

Archer

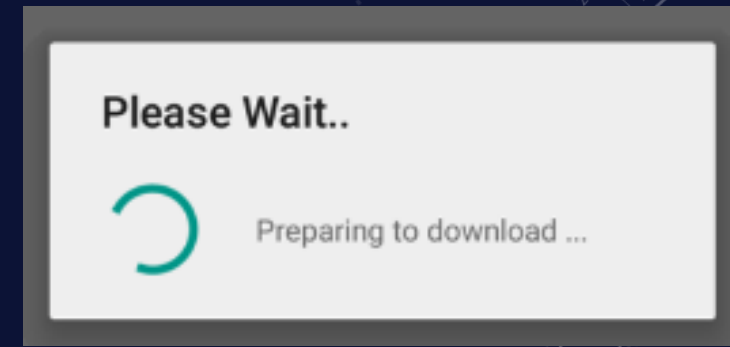
Wizard

Boîtes de dialogues

ProgressDialog

Un **ProgressDialog** est une extension du **AlertDialog** qui a pour principal but d'afficher l'avancement d'une tâche. Il contient essentiellement le widget **ProgressBar**, en plus d'un titre et d'un message.

La méthode **setCancelable(true)** rend la boîte de dialogue annulable et **setIndeterminate(true)** permet d'ignorer la progression et la boîte de dialogue affiche une animation infinie à la place

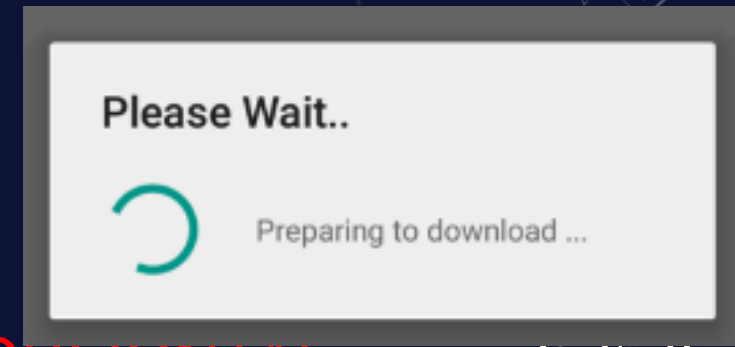


Boîtes de dialogues

ProgressDialog

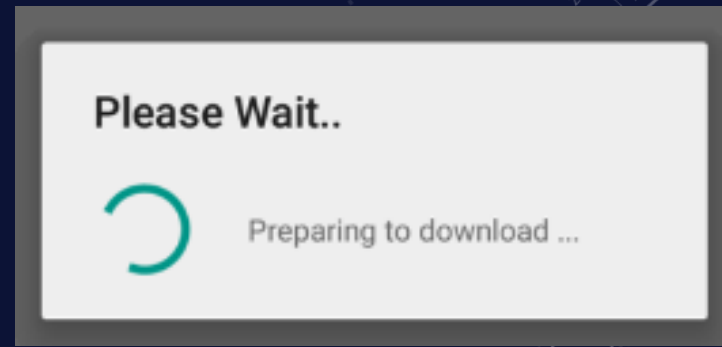
Un **ProgressDialog** possède 2 styles : **STYLE_HORIZONTAL** permet d'afficher un **ProgressBar** horizontal tandis que **STYLE_SPINNER** permet de montrer un **ProgressBar** circulaire dans la boîte de dialogue.

Pour appliquer un style dans un **ProgressDialog**, il faut utiliser la méthode **setProgressStyle(...)**, qui prend comme valeur par défaut **STYLE_SPINNER**.



Boites de dialogues

ProgressDialog

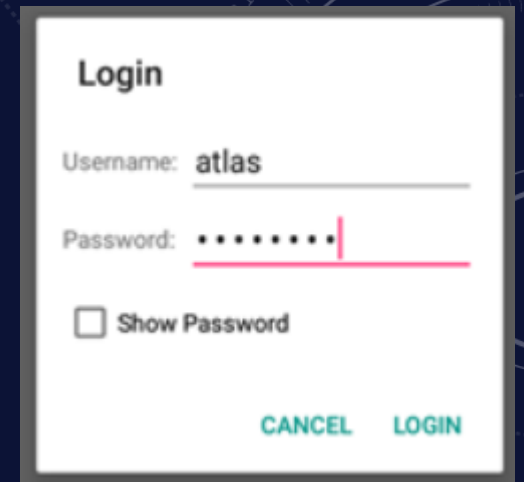


```
ProgressDialog dialog = new ProgressDialog(this);  
dialog.setCancelable(true);  
    dialog.setIndeterminate(true);  
    dialog.setTitle("Please Wait...");  
    dialog.setMessage("Preparing to download...");  
    dialog.setProgressStyle(ProgressDialog.STYLE_SPINNER);  
    dialog.show();
```

Boîtes de dialogues

Boîtes de dialogue personnalisées

Pour des raisons d'ergonomies et de spécificités, il peut être intéressant de créer une boîte de dialogue personnalisée, en y ajoutant des widgets, par exemples des `EditText` et des `TextView`. Pour ce faire, il faut étendre de la classe `Dialog`.



The image shows a custom login dialog box with a white background and a gray border. It contains the following elements:

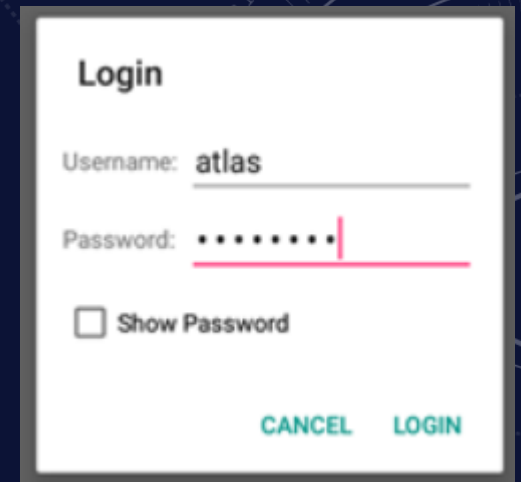
- Title:** "Login" in bold black text at the top left.
- Username field:** Labeled "Username:" followed by a text input field containing the text "atlas".
- Password field:** Labeled "Password:" followed by a text input field with masked characters (dots) and a red vertical cursor line on the right.
- Show Password checkbox:** A small square checkbox followed by the text "Show Password".
- Buttons:** Two buttons at the bottom right, labeled "CANCEL" and "LOGIN" in green capital letters.

Boîtes de dialogues

Boîtes de dialogue personnalisées

Création de la boîte de dialogue

Pour commencer, il faut créer un nouveau **layout** (par exemple, **dialog_custom.xml**), puis y construire la vue avec la disposition des widgets quand souhaite afficher dans la boîte de dialogue personnalisée.



Login

Username: atlas

Password: |

☐ Show Password

CANCEL LOGIN

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

Boîtes de dialogues

Boîtes de dialogue personnalisées

Maintenant, si la boîte de dialogue s'affiche pour la première fois, la méthode `onCreate(...)` est appelée, c'est dans cette méthode que la boîte de dialogue est instanciée. Pour affecter le `layout` précédemment créé, il suffit de l'associer à la boîte de dialogue en utilisant la méthode `setContentView(...)`



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

Boîtes de dialogues

Boîtes de dialogue personnalisées

```
public class CustomDialog extends Dialog {  
  
    public CustomDialog(Context context){  
        super(context);  
        ...  
    }  
    @Override  
    protected void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.dialog_detail);  
        ...  
    }  
}
```

A login dialog box with a white background and a gray border. It contains a title "Login", a "Username:" label followed by a text input field containing "atlas", a "Password:" label followed by a password input field with masked characters ".....", and a checkbox labeled "Show Password".

Login

Username: atlas

Password:

☐ Show Password

Boîtes de dialogues

Boîtes de dialogue personnalisées

- Affichage et fermeture de la boîte de dialogue

Après avoir créé l'objet de la boîte de dialogue personnalisée, il est possible de l'afficher en utilisant la méthode **show()** et la fermer en appelant la méthode **dismiss()**.



The image shows a login dialog box with a white background and a gray border. At the top, the title "Login" is displayed. Below it, there are two input fields: "Username:" with the text "atlas" and "Password:" with masked characters ".....". A checkbox labeled "Show Password" is located below the password field. At the bottom right, there are two buttons: "CANCEL" and "LOGIN".

```
CustomDialog dialog = new CustomDialog(MainActivity.this);  
dialog.show();  
...  
dialog.dismiss();
```


LIENS UTILES

Les étudiants peuvent consulter ces références pour approfondir leurs connaissances :

✓ AdapterView :

<https://developer.android.com/reference/android/widget/AdapterView.html>

✓ ListView à adaptateurs personnalisés :

<http://www.journaldev.com/10416/android-listview-with-custom-adapter-example-tutorial>

✓ ProgressDialog avancé :

<http://www.oodelestechnologies.com/blogs/Custom-Progressbar-and-ProgressDialog>

✓ Boite de dialogue personnalisée :

<http://www.codexpedia.com/android/android-custom-dialog-example/>

✓ Débogage sous Android studio :

<https://www.learnhowtoprogram.com/android/user-interface-basics-637d41b1-35dc-400a-bcc3-65794760474d/debugging-breakpoints-and-the-android-debugger>