



# Android

" Cycle Développeur "

# Dans ce document

**Objectif du document:**

**Ce document est composé des parties suivantes:**

1. Présentation d'Android.
2. Développement Android.
3. Les librairies
4. IHM.
5. Layout et Composants
6. Le modèle de composants.
7. La persistance de données.
8. Gestion du réseau.
9. Complément.



# Présentation d'Android

" Cycle Developpeur "

# Android



- Android est un système d'exploitation **fondé en 2003** par Android Corporated. Racheté en 2005 par Google.
- Destiné aux téléphones portables, tablettes et autres périphériques (ex: baladeurs, appareils photos, autoradio, etc).
- Il peut accueillir des applications tierces.
- La majeure partie des applications est distribuée via l'Android Play Store, qui contient plus de 2 000 000 applications en Février 2016 (Selon le site [www.statista.com](http://www.statista.com)).



# Philosophie



- Android est sous licence Open Source.
- Android est gratuit.
- Android est facile à développer.
- Les applications Android sont valorisées via le Play Store.
- Android est flexible.
- Android est construit sur un principe d'application.

# Problématiques



Les problématiques sont liées principalement aux problématiques du développement sous mobiles :

- Ressources matérielles limitées.
- Interaction avec le système complet du périphérique sans interruption.
- Taille des écrans réduits.
- IHM à utilisation tactile.
- Internationalisation (i18n)
- Composants matériels différents ou absents.
- Version

# Kernel



- Basé sur un **kernel linux**
- Ne possède pas de système de fenêtrage natif.
- **Glibc** non supporté.
- **Bionic libc** est utilisé.
- Différents **patches** sont ajoutés pour la gestion d'alimentation, le partage de mémoire, etc. pour une meilleure gestion des caractéristiques d'un appareil mobile.

# SDK Software Development Kit



- SDK Android, le kit de développement d'Android, est un ensemble complet d'outils de développement.
- Il inclut :
  - Un débogueur
  - Des bibliothèques logicielles
  - Un émulateur basé sur QEMU
  - De la documentation
  - Des exemples de code
  - Des tutoriaux.
- Pour gérer ces outils, on utilise le **SDK Manager**.

# Android Studio



**Android Studio** est un environnement de développement (IDE) permettant de développer des applications Android. Il est basé sur IntelliJ IDEA.

## Historique

Avant Android Studio, de 2004 à 2014, Google proposait comme IDE officiel une distribution spécifique de l'Eclipse, contenant le SDK d'Android.

Le 8 Décembre 2014, Android Studio passe de version bêta à stable 1.0. L'IDE devient alors conseillé par Google.

## Fonctionnalité

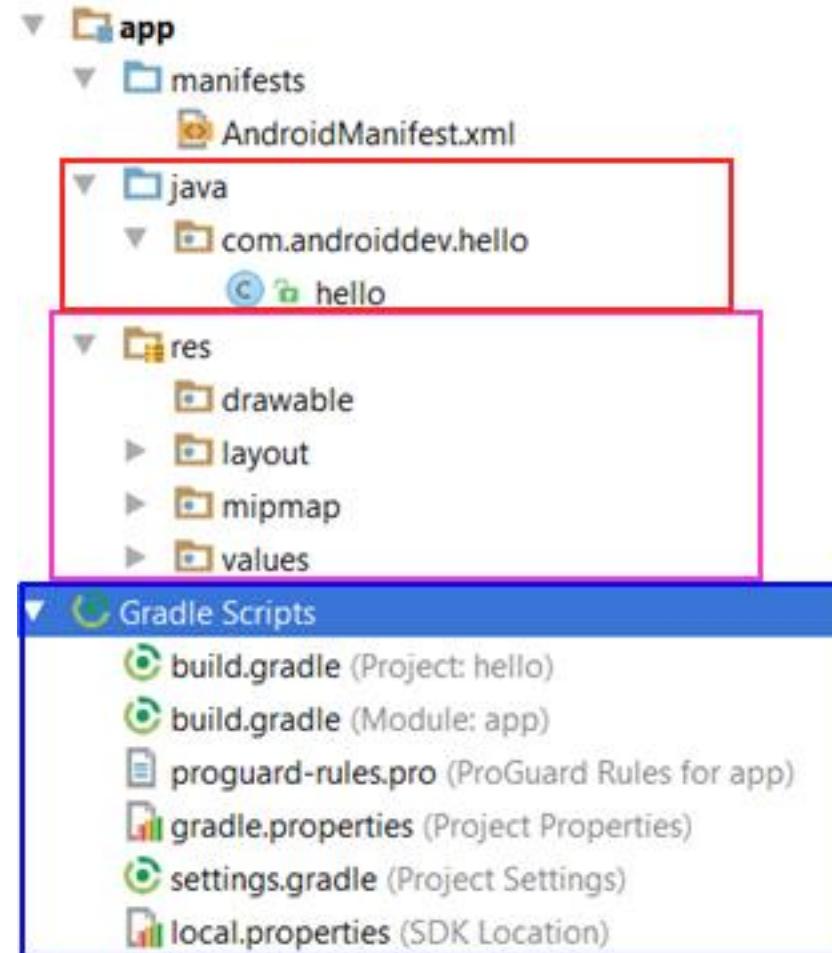
Android Studio permet principalement d'éditer les fichiers Java et les fichiers de configuration d'une application Android.

Il propose, entre autres, des outils pour gérer le développement d'applications multilingues et permet de visualiser la mise en page des écrans sur des écrans de résolutions variées simultanément.

# Structure Android Studio



## Arborescence



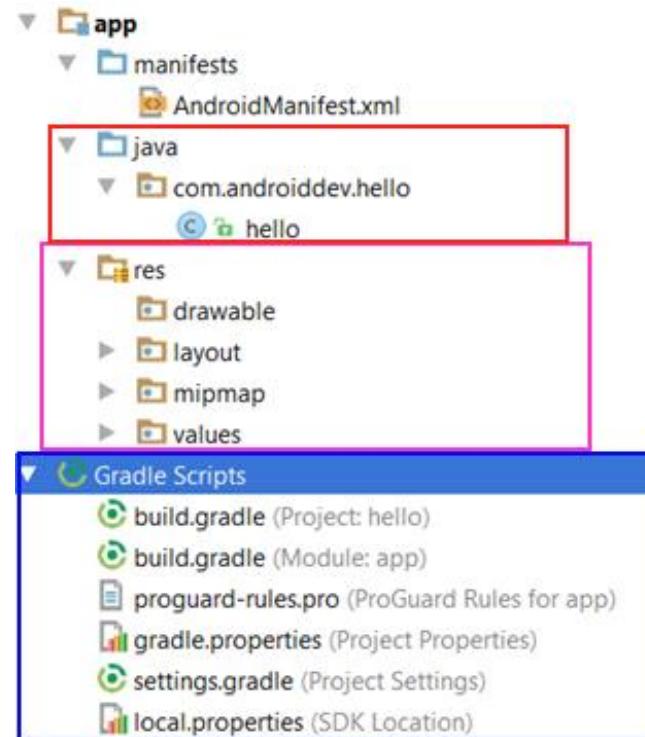
# Eclipse à Android Studio



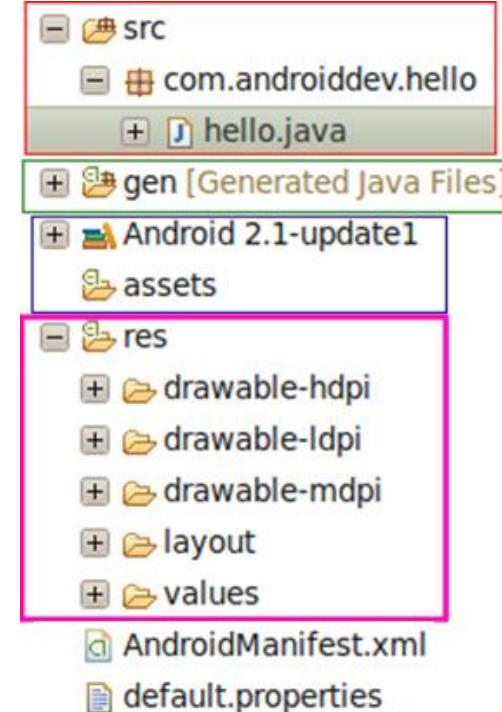
Les projets Eclipse et Android Studio sont bien différents. Cependant, on retrouve une arborescence quasi-similaire :

## Arborescence

### ➤ Android Studio



### ➤ Eclipse ADT



Gradle est le digne successeur de Maven et de Apache Ant, alliant ces deux outils afin de créer une plateforme de production Java simple à utiliser et bien adaptée pour les projets Android.

- Intégré dans Android Studio, il gère et construit les projets Android (en utilisant le langage Groovy).
- Permet de gérer la construction d'un projet multi-modules et dépendances Maven de manière très simple.

## **setting.gradle**

Ce fichier, placé à la racine, permet de référencer la liste des modules que Gradle doit gérer.

## **build.gradle**

- Le fichier build.gradle, placé à la racine, contient des informations partagées pour tous les modules.
- Celui placé dans chaque module contient des informations propres à chaque module.

# Android Studio vs Eclipse

La question qui revient toujours est :

Quelle est la grande différence entre Android Studio et Eclipse ADT?

	<b>Android Studio</b>	<b>Eclipse ADT</b>
Système de compilation	<b>Gradle</b>	<b>Ant</b>
Dépendances sur la base Maven	Oui	Non
Variations de compilation et generation multiple d'APK	Oui	Non
Advanced Android code completion and refactoring	Oui	Non
Editeur de vue graphique	Oui	Oui
APK signée et manager de keystore	Oui	Oui
Support NDK	Oui	Oui

# Pré requis



Pour commencer à développer sur la plateforme Android, il faut avoir un environnement JAVA Développeur (JDK). Il faudra donc, avant de commencer, récupérer les paquets suivants :

## **Java Development Kit (JDK 8 conseillé sur le site officiel)**

JDK est un ensemble de bibliothèques logicielles de base de JAVA.

## **Android Studio**

IDE conseillé par Google.



# Développement Android

" Cycle Developpeur "

# La base d'une application



Android Studio nous permet de créer des applications modulaires (contenant plusieurs modules)

Il y aura donc une notion de projet et de modules.

## **Le projet:**

Le projet est le répertoire racine de l'application, celui qui regroupera tous les modules et configurations de notre application d'ordre général.

## **Les modules:**

Les modules sont des sous projets pouvant être utilisés en tant que librairie ou module de lancement.

Un module de lancement peut être spécifique à un type périphérique (Smartphone, tablette, montre, etc.).

# Le AndroidManifest.xml



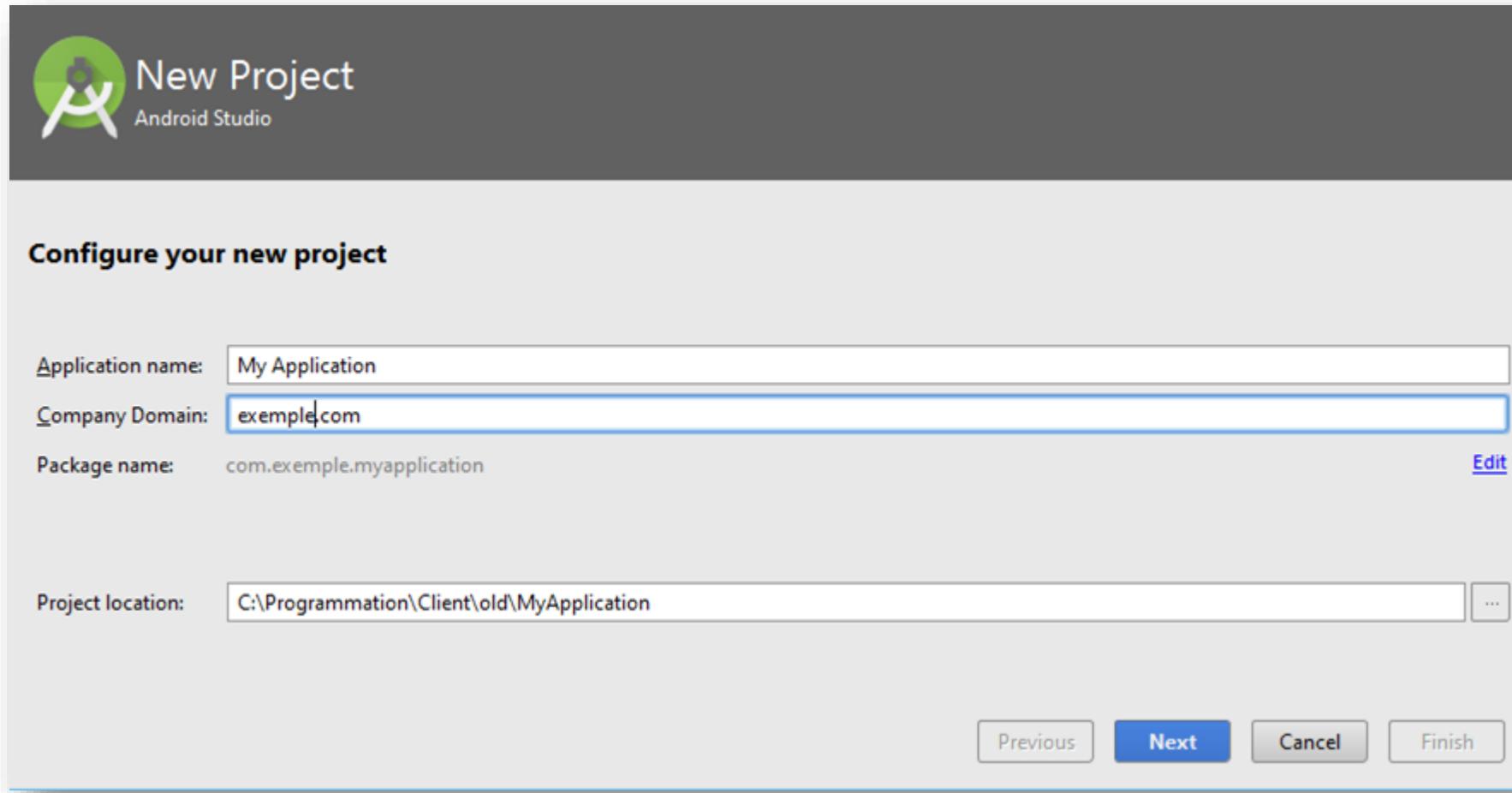
Toutes les applications ont un *AndroidManifest.xml* placé à son répertoire racine.

Ce fichier présente les informations essentielles sur l'application. Le système doit avoir ces informations avant d'exécuter un des codes de l'application.

## Fonction:

- Nommer le paquet Java pour l'application. Le nom du package est utilisé comme **identifiant unique** pour l'application.
- Décrit les composants de l'application (les activités, les services, les récepteurs de radiodiffusion, les fournisseurs de contenu dont l'application est composée).
- Détermine quel processus sera l'hôte des composants.
- Déclare les **autorisations nécessaires** pour interagir avec les composantes de l'application.

# Création de projet



# Création de projet



Phone and Tablet

Minimum SDK API 23: Android 6.0 (Marshmallow) ▾

Lower API levels target more devices, but have fewer features available.  
By targeting API 23 and later, your app will run on approximately **4.7%** of the devices  
that are active on the Google Play Store.  
[Help me choose](#)

Wear

Minimum SDK API 21: Android 5.0 (Lollipop) ▾

TV

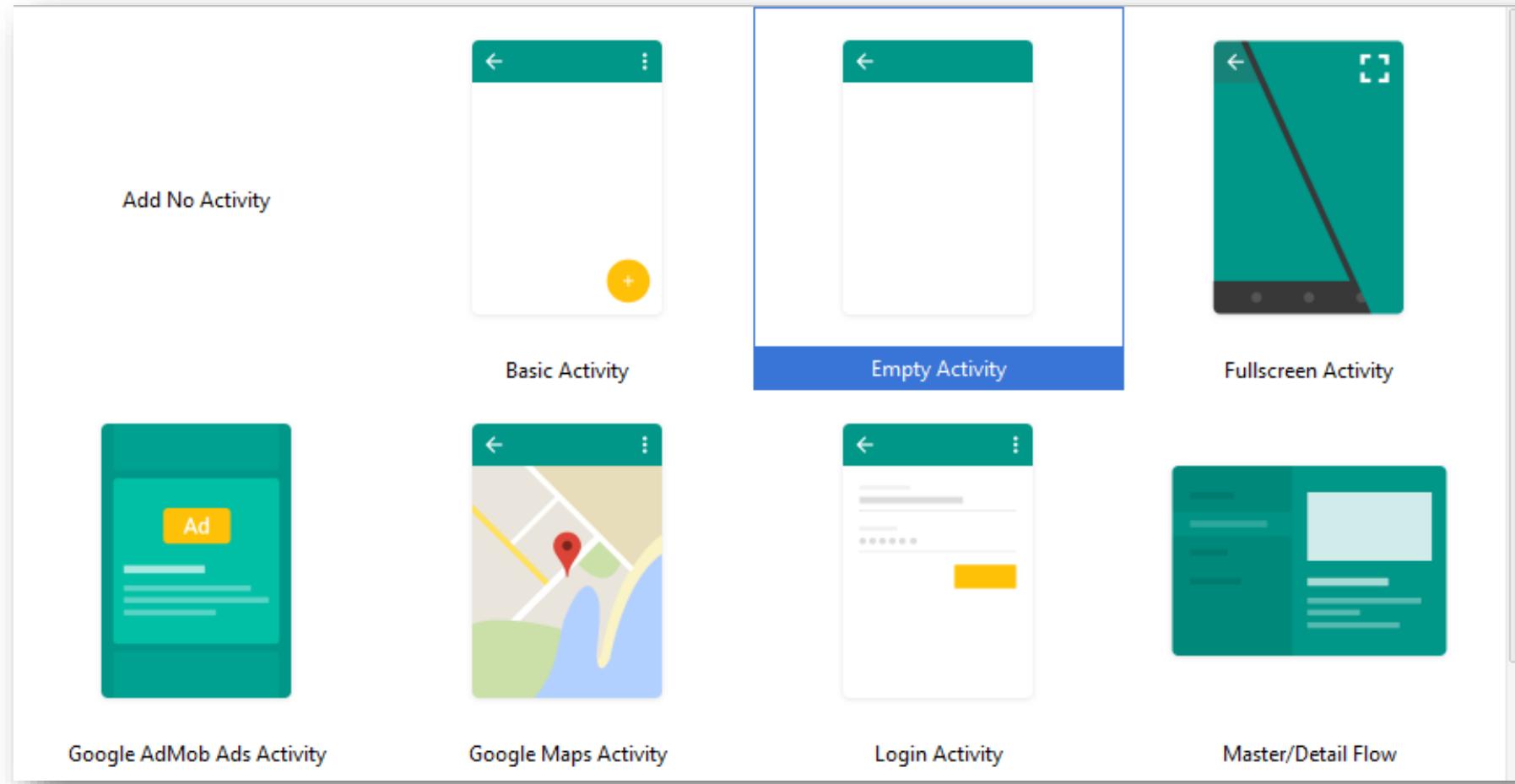
Minimum SDK API 21: Android 5.0 (Lollipop) ▾

Android Auto

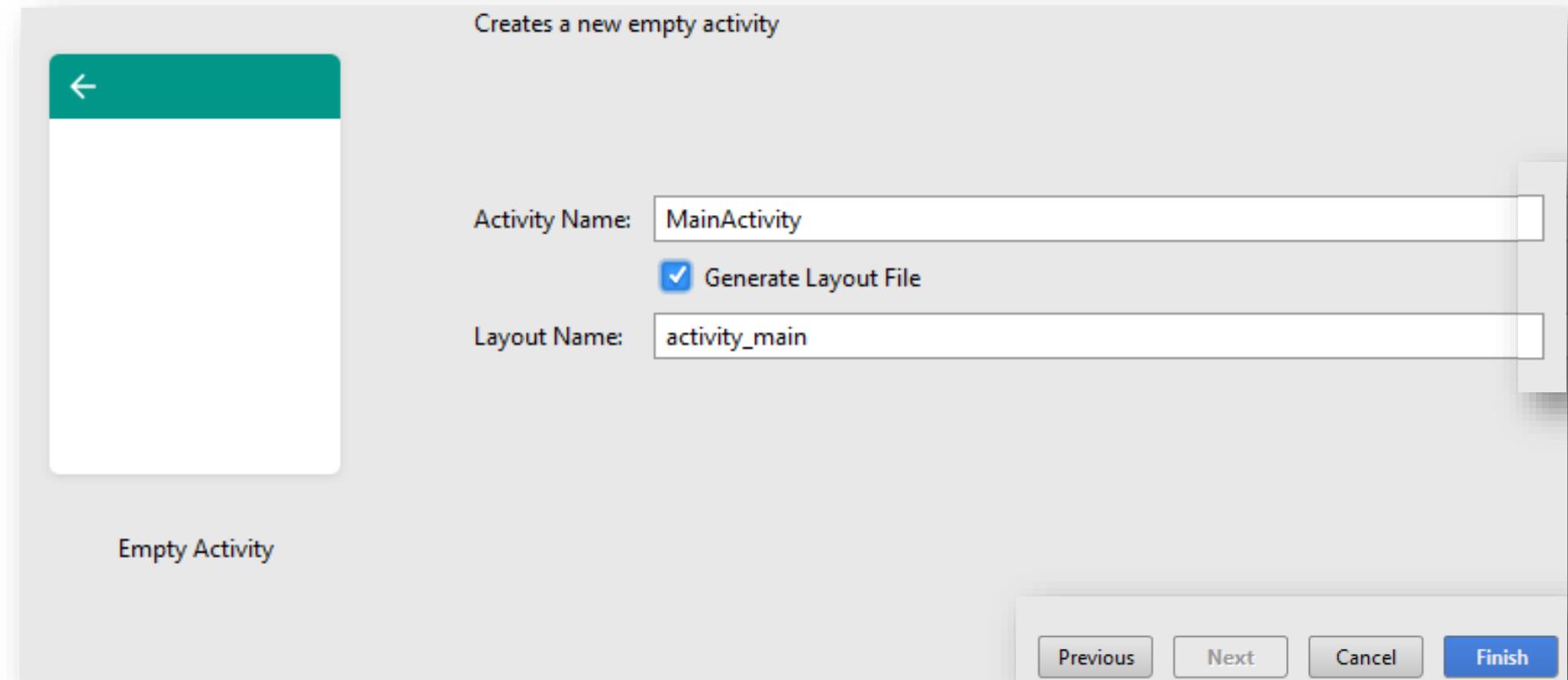
Glass

Minimum SDK Glass Development Kit Preview (API 19) ▾

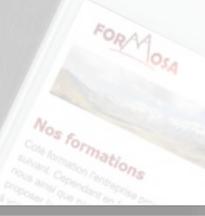
# Création de projet



# Création de projet



# Prise en main



The screenshot shows the Android Studio interface with the following details:

- Toolbar:** Standard Android Studio icons for file operations, navigation, and build.
- Project Navigational Bar:** Shows the current project path: `MyApplication > app > src > main > java > com > exemple > myapplication > MainActivity`.
- Project Structure View:** On the left, it shows the project structure under "1: Project". It includes:
  - app:** Contains `manifests`, `java` (which contains `com.exemple.myapplication`), `res`, and `Gradle Scripts`.
    - `MainActivity` is selected in the `java` folder.
  - Captures:** A section for screenshots.
- Main Activity Code:** The `MainActivity.java` file is open in the editor. The code is as follows:

```
1 package com.exemple.myapplication;
2
3 import ...
4
5
6 public class MainActivity extends AppCompatActivity {
7
8     @Override
9     protected void onCreate(Bundle savedInstanceState) {
10         super.onCreate(savedInstanceState);
11     }
12 }
13 }
```

A yellow callout box points to the `onCreate` method, indicating it is the target for further discussion.

# Prise en main



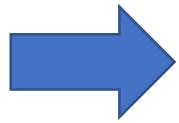
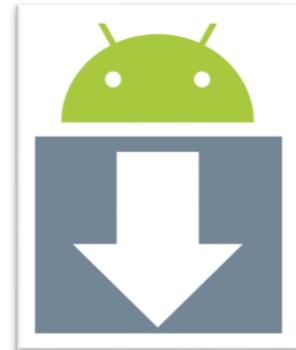
The screenshot shows the Android Studio interface. On the left, the Project tool window displays the structure of the 'MyApplication' project located at C:\Programmation\Client\old\MyApplication. The 'app' module is selected. Other files visible include .gradle, .idea, .gitignore, build.gradle, gradle.properties, gradlew, gradlew.bat, local.properties, MyApplication.iml, settings.gradle, and External Libraries. On the right, the code editor shows the 'MainActivity.java' file:

```
1 package com.example..  
2  
3 import ...  
4  
5  
6 public class MainAct...  
7  
8     @Override  
9     protected void onCrea...  
10    super.onCreate(...)  
11 }  
12 }  
13 }
```

# SDK Manager



Nos formations  
Code formation l'entreprise  
Suivant... Capacitant... nos équipes  
nos équipes... proposent...  
FOR MOSA



Android SDK Location: D:\SdkManager [Edit](#)

SDK Platforms [SDK Tools](#) [SDK Update Sites](#)

Each Android SDK Platform package includes the Android platform and sources pertaining to an API level by default. Once installed, Android Studio will automatically check for updates. Check "show package details" to display individual SDK components.

	Name	API Level	Revision	Status
<input checked="" type="checkbox"/>	Android N Preview	N	2	Partially installed
<input checked="" type="checkbox"/>	Android 6.0 (Marshmallow)	23	3	Installed
<input checked="" type="checkbox"/>	Android MNC Preview	MNC	1	Installed
<input checked="" type="checkbox"/>	Android 5.1 (Lollipop)	22	2	Installed
<input checked="" type="checkbox"/>	Android 5.0 (Lollipop)	21	2	Installed
<input checked="" type="checkbox"/>	Android 4.4 (KitKat Wear)	20	2	Installed
<input checked="" type="checkbox"/>	Android 4.4 (KitKat)	19	4	Installed
<input checked="" type="checkbox"/>	Android 4.3 (Jelly Bean)	18	3	Installed
<input checked="" type="checkbox"/>	Android 4.2 (Jelly Bean)	17	3	Installed
<input checked="" type="checkbox"/>	Android 4.1 (Jelly Bean)	16	5	Installed
<input checked="" type="checkbox"/>	Android 4.0.3 (IceCreamSandwich)	15	5	Installed
<input checked="" type="checkbox"/>	Android 4.0 (IceCreamSandwich)	14	4	Installed
<input checked="" type="checkbox"/>	Android 3.2 (Honeycomb)	13	1	Installed
<input checked="" type="checkbox"/>	Android 3.1 (Honeycomb)	12	3	Installed
<input checked="" type="checkbox"/>	Android 3.0 (Honeycomb)	11	2	Installed
<input checked="" type="checkbox"/>	Android 2.3.3 (Gingerbread)	10	2	Installed
<input checked="" type="checkbox"/>	Android 2.2 (Froyo)	8	3	Installed
<input checked="" type="checkbox"/>	Android 2.1 (Eclair)	7	3	Installed
<input checked="" type="checkbox"/>	Android 1.6 (Donut)	4	3	Installed
<input checked="" type="checkbox"/>	Android 1.5 (Cupcake)	3	4	Installed

Show Package Details

# Le manifest



Ce fichier est indispensable pour tous les projets Android. C'est un fichier XML qui décrit l'ensemble des éléments utilisés dans l'application.

Il est composé de plusieurs nœuds:

- **manifest** : Le nœud root.
- **uses-sdk** : permet de filtrer les périphériques sur lesquels l'application pourra être exécutée.
- **uses-feature** : permet de définir les périphériques autorisés à exécuter l'application.
- **application** : décrit les attributs qui caractérisent l' application et en énumère les composants. Par défaut, votre application n'a qu'un composant, l'activité principale.

# Le manifest



- **activity** : permet de décrire les activités utilisées dans l'application.
- **intent-filter/action** : indique quelle est l'activité à lancer.
- **intent-filter/category** : indique quelle est l'activité à lancer.
- **uses-permission** : Pour demander l'autorisation d'accéder à certaines ressources.
- **permission** : Pour définir ses propres permissions d'accès à l'application.

# Log



Pour afficher des messages dans le moniteur, on utilise les méthodes:

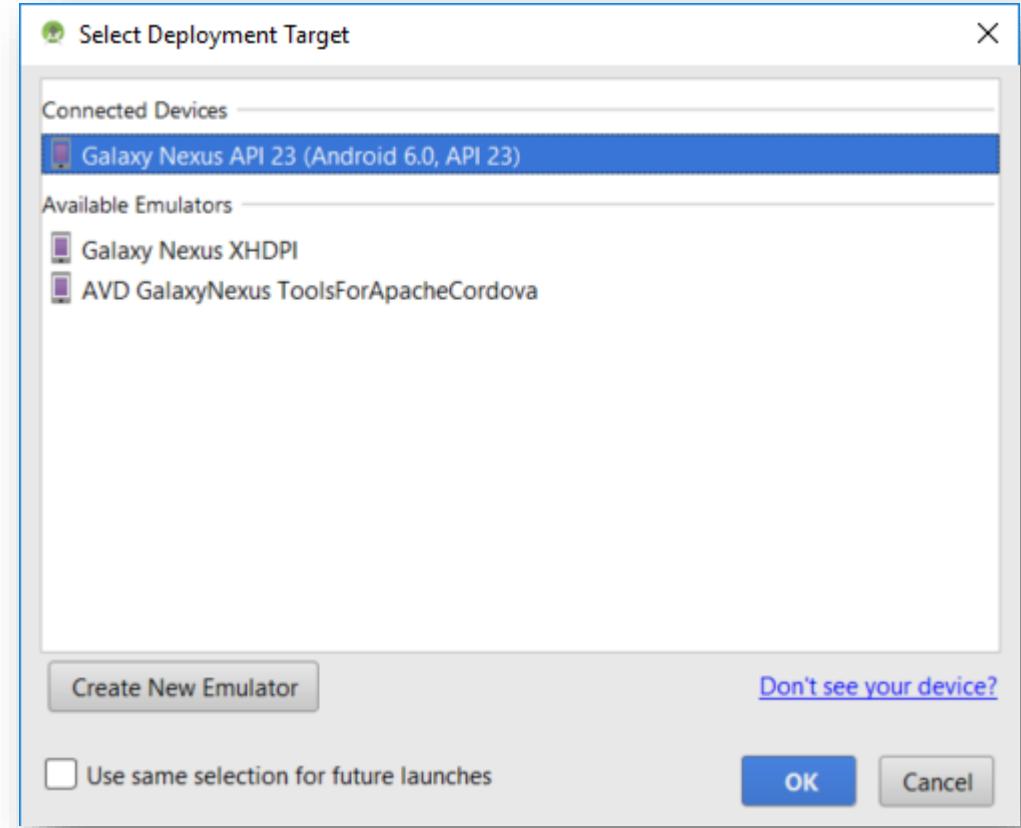
Log.v(), Log.i(), Log.e(), Log.wtf(), Log.d(), Log.w().

## Explication:

- Log.v(): VERBOSE
- Log.d(): DEBUG
- Log.i(): INFORMATION
- Log.w(): WARNING
- Log.e(): ERROR
- Log.wtf(): ASSERT

# Déployer sur un émulateur

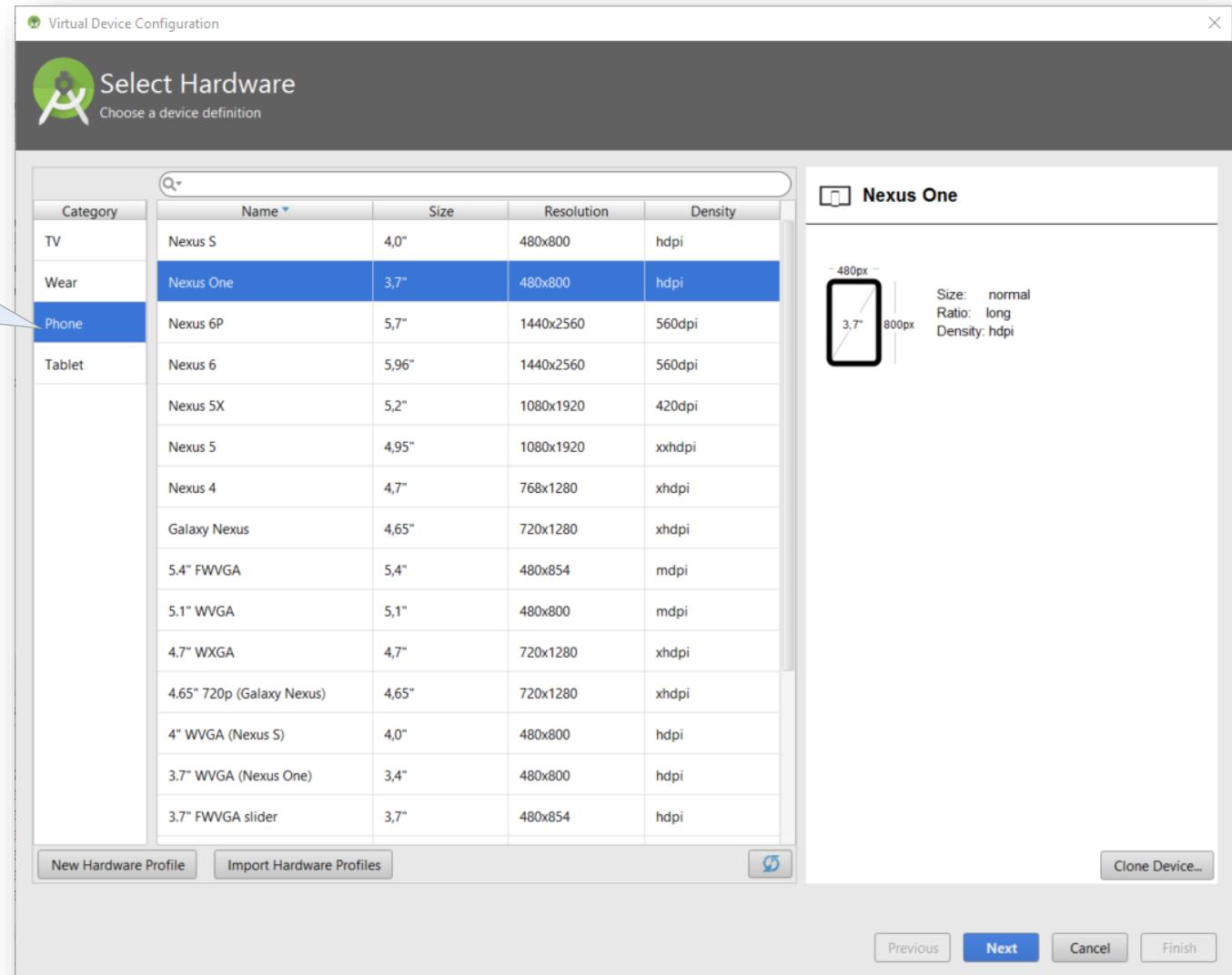
La création d'un émulateur va se faire lors du démarrage de l'application en cliquant sur le bouton « Create New Emulator ».



# Déployer sur un émulateur

Le choix du Hardware où il est question de choisir le téléphone à émuler.

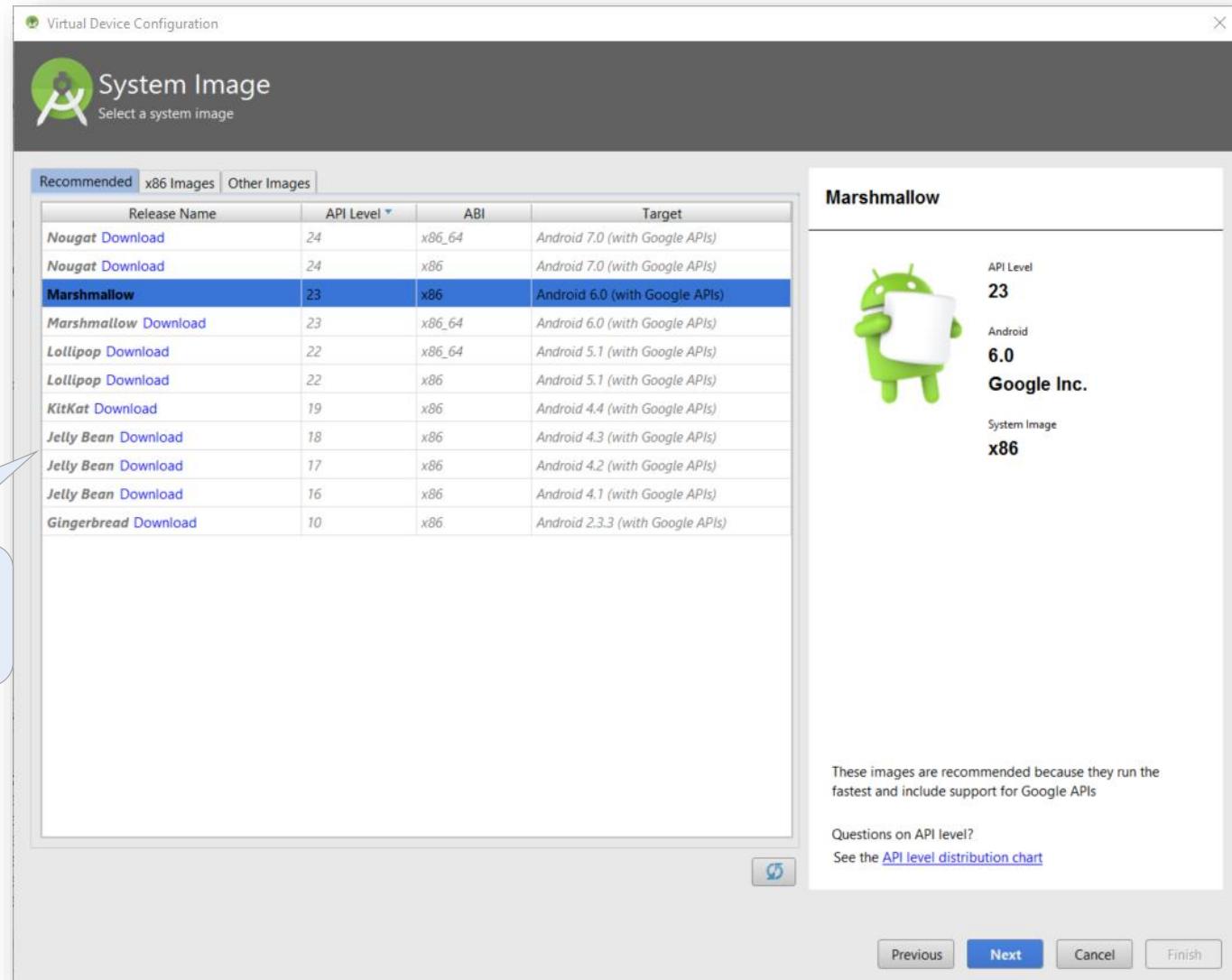
Choix de la catégorie et du type de téléphone



# Déployer sur un émulateur

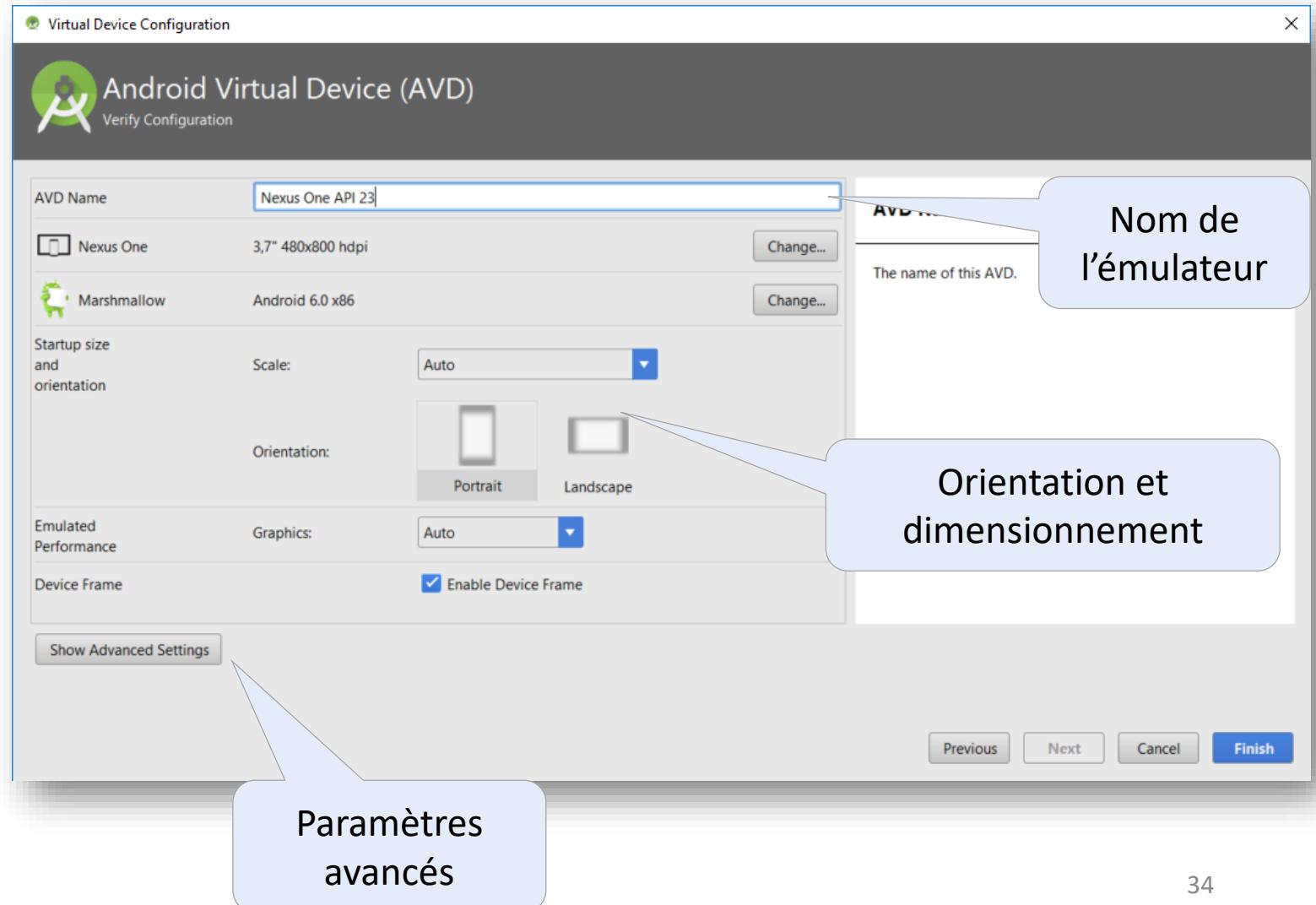
Le choix de l'image du Système proposant  
Les différentes version d'Android.

Choix de l'image du  
système



# Déployer sur un émulateur

La validation et les dernières configurations pour l'émulateur

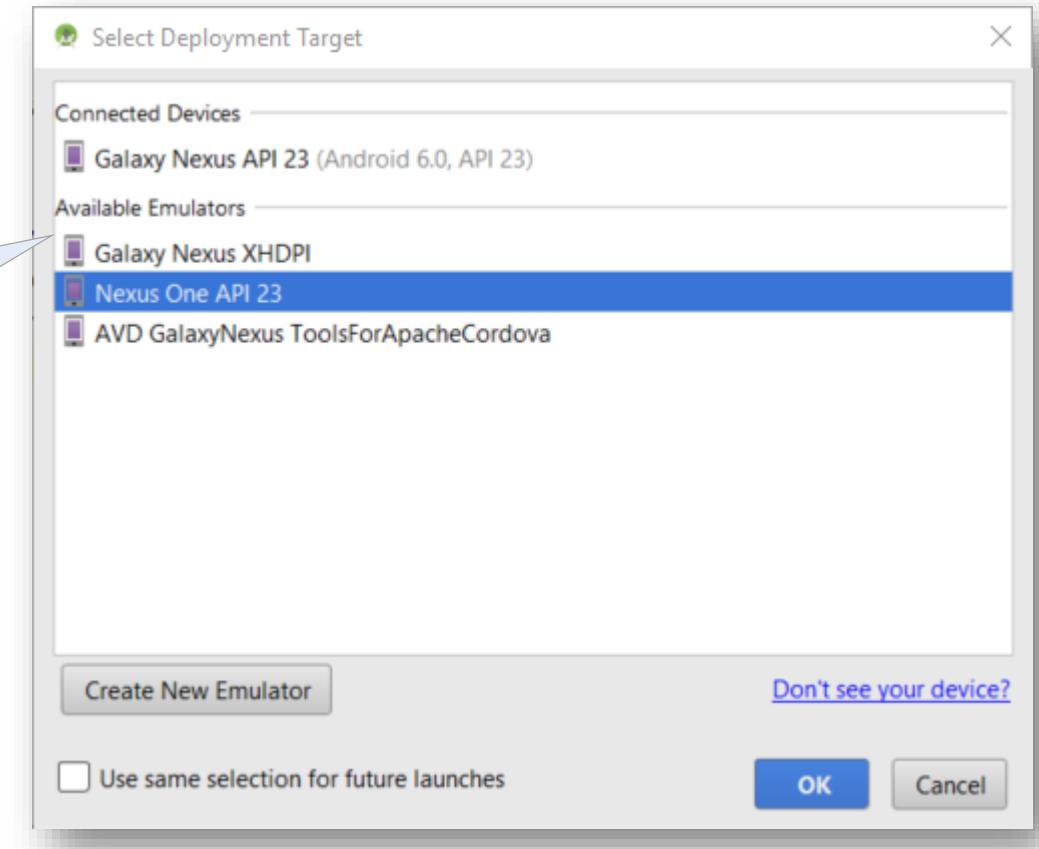


# Démarrer l'application sur une activité

L'émulateur apparaît alors dans la liste des choix au démarrage de l'application.

Choix de l'émulateur

Au final il suffit de démarrer l'émulateur créé



# Démarrer une autre activité



Une application Android se compose de plusieurs activités.

L'activité se créant au lancement d'une application se configure dans le manifest

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.axopen.tpl">
    <application
        android:allowBackup="true"
        android:icon="@mipmap/ic_launcher"
        android:label="@string/app_name"
        android:supportsRtl="true"
        android:theme="@style/AppTheme">
        <activity android:name=".MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Configuration de l'activité au démarrage de l'application



# Les Bibliothèques

" Cycle Developpeur "

# Mécanisme



Sur Android Studio, il y a deux catégories de librairies.

- Dépendances
- Natives

Il y a 3 catégories de librairies dites dépendances :

- Gradle
- Archive
- Module

# Les dépendances



## Gradle

Les librairies gradle compatibles à Android sont disponibles sur Android Studio via son interface d'import. Pour retrouver une librairie manuellement, il faut saisir dans le fichier "build.gradle" dans "dependencies" :

```
compile '<GroupId>:<ArtifactId>:<LastVersion>'
```

## Archive

Les librairies archives sont en **.jar** (format d'archive utilisé par Java) ou en **.aar** (distribution binaire d'un projet de bibliothèque Android). Pour les utiliser, elles doivent être placées dans un dossier du projet.

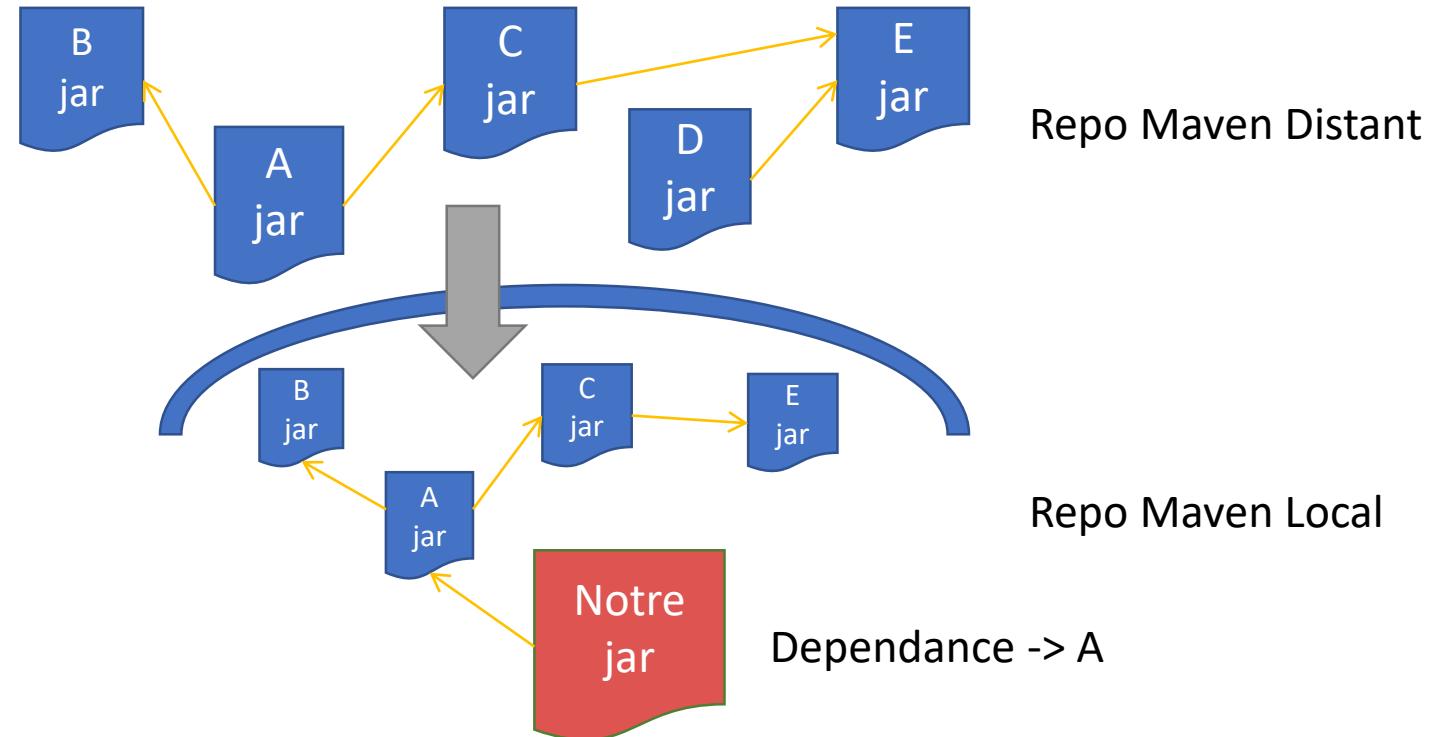
## Module

Sur Android Studio, il est possible d'avoir un projet contenant plusieurs modules. Ces modules, peuvent être indépendants mais peuvent aussi dépendre les uns des autres.

# Apache Maven

Apache Maven est un « repository » contenant une liste de bibliothèques avec les liens de dépendances.

Ainsi lorsque nous créons notre programme la déclaration des dépendances va « tirer » toutes les autres dépendances.



# Dépendances



The screenshot shows the 'Dependencies' tab in the Android Studio project settings. On the left, a sidebar lists various project components: SDK Location, Project, Developer Services, Ads, Analytics, Authentication, Cloud, Notifications, Modules, and app. The 'app' module is currently selected, indicated by a blue background. The main area displays the project's dependency tree. At the top, there is a placeholder entry: '{dir=libs, include=[\*.jar]}'. Below it, four dependencies are listed, all with a scope of 'Compile':

Dependency	Scope
com.android.support:appcompat-v7:22.2.0	Compile
libs/touroparc-lib.jar	Compile
com.google.code.gson:gson:2.3.1	Compile
mysql:mysql-connector-java:5.1.36	Compile

## . AAR Format

Le bundle 'aar' est la distribution binaire d'un projet de bibliothèque Android.

L'extension du fichier est .aar, le type de la structure est .aar, mais le fichier lui-même est un fichier zip simple avec les entrées suivantes:

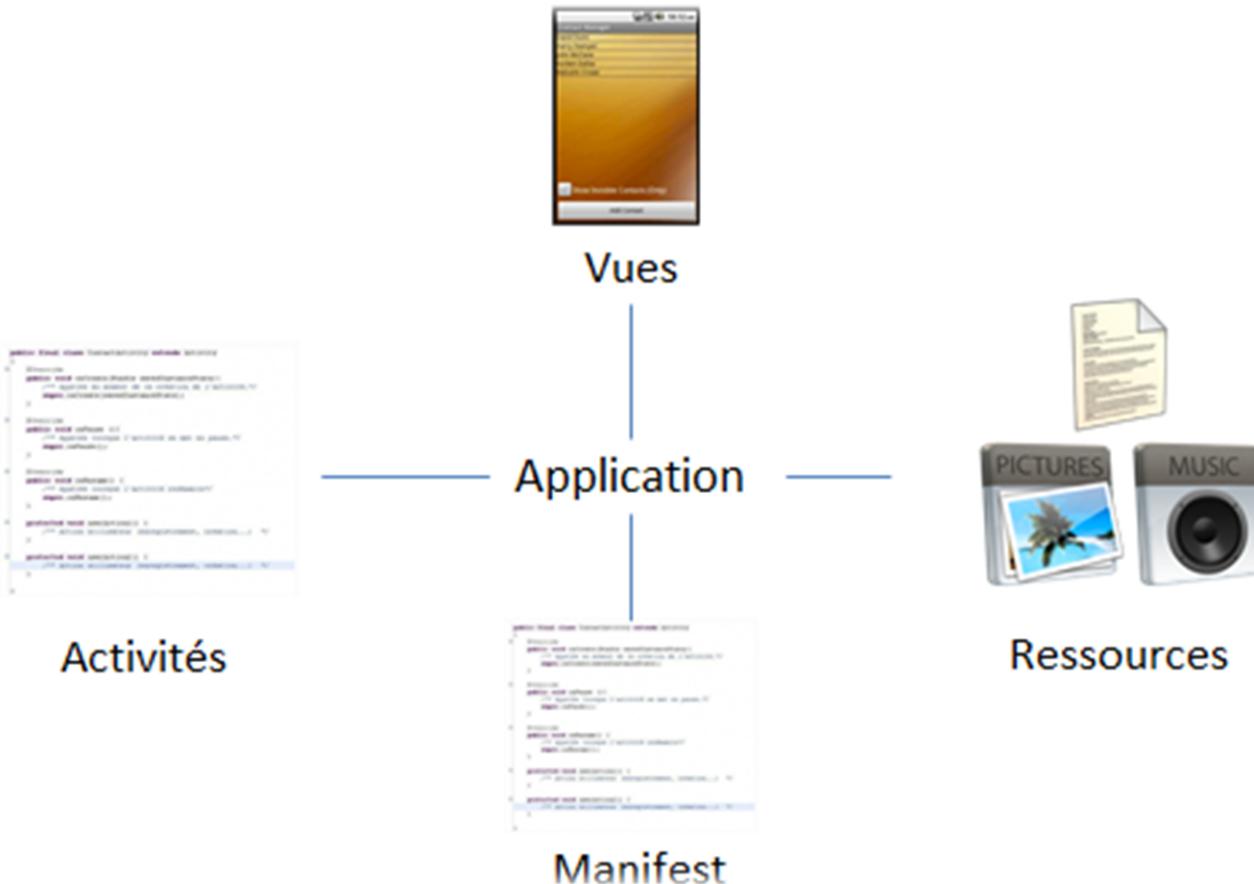
- /AndroidManifest.xml (obligatoire)
- /classes.jar (obligatoire)
- /res/ (obligatoire)
- /R.txt (obligatoire)
- /assets/ (optional)
- /libs/\*.jar (optional)
- /jni/<abi>/\*.so (optional)
- /proguard.txt (optional)
- /lint.jar (optional)



# Les IHM

" Cycle Developpeur "

# Vue d'ensemble



# Ressources



- Images, sons, chaînes de caractères, styles, etc.
- Toutes les ressources sont dans **/res**
- Organiser de manière précise, permet de répondre aux problématiques des systèmes embarqués :
  - Pouvoir être exécutés sur différents périphériques.
  - Proposer des applications multilingues.

# Type de ressources

- res/drawable
- res/layout
- res/menu
- res/raw
- res/values
- res/mipmap



# Aspect général



```
<?xml version= "1.0" encoding="utf-8"?>
<resources>
    ...
    <type_ressource name="clé_ressource">valeur</type_ressource >
</resources>
```

## Référence :

Lorsqu'une ressource a un identifiant, l'IDE ajoute au fichier R.java une référence à l'identifiant de cette ressource.

La syntaxe pour récupérer cette référence depuis du code java:

**R.type\_de\_ressource.nom\_de\_la\_ressource**

# Internationalisation (i18n)



Android est capable de retrouver automatiquement des ressources car elles sont toutes référencées dans un fichier.

Pour les strings, un éditeur a été mis en place pour faciliter l'ajout et la traduction de valeur.

<input type="checkbox"/> Show only keys needing translations	<a href="#">Order a translation...</a>			
Key	Default Value	Untranslatable	French (fr)	Bafia (ksf)
action_settings	Settings	<input type="checkbox"/>	Paramètre	Settings
app_name	AS Buers Basket	<input type="checkbox"/>	AS Buers Basket	AS Buers Basket
title_activity_activity_le_club	The club	<input type="checkbox"/>	Le club	

# Les chaines de caractères



Le fichier les fichiers « string.xml » sont utilisés par l'éditeur pour enregistrer les valeurs en fonction d'une clé.

La saisie manuelle peut se faire en respectant cette syntaxe :

```
<string name="nomChaîne">Le texte de la chaîne s\'appelle \"nomChaîne\"</string>
```

## Remarque :

Il faut penser à échapper les guillemets et les apostrophes, sauf si la chaîne est encadrée d'apostrophe. Dans ce cas, on échappera uniquement les apostrophes.

# Les drawables



Les **drawables** sont des images.

Ces images peuvent être de **2 types** :

- Matricielles : Android supportant le PNG, le GIF et le JPEG (recommandé).
- Extensibles : ce sont des images qu'Android est capable d'étirer lui-même.

Le nom du fichier déterminera **l'identifiant** du drawable.

Il peut contenir toutes les lettres minuscules, tous les chiffres et des underscores (\_), mais **pas de majuscules**.

On pourra récupérer le drawable à l'aide de **R.drawable.nom\_du\_fichier\_sans\_l\_extension**

# Les styles



Les styles permettent de définir une **charte graphique**.

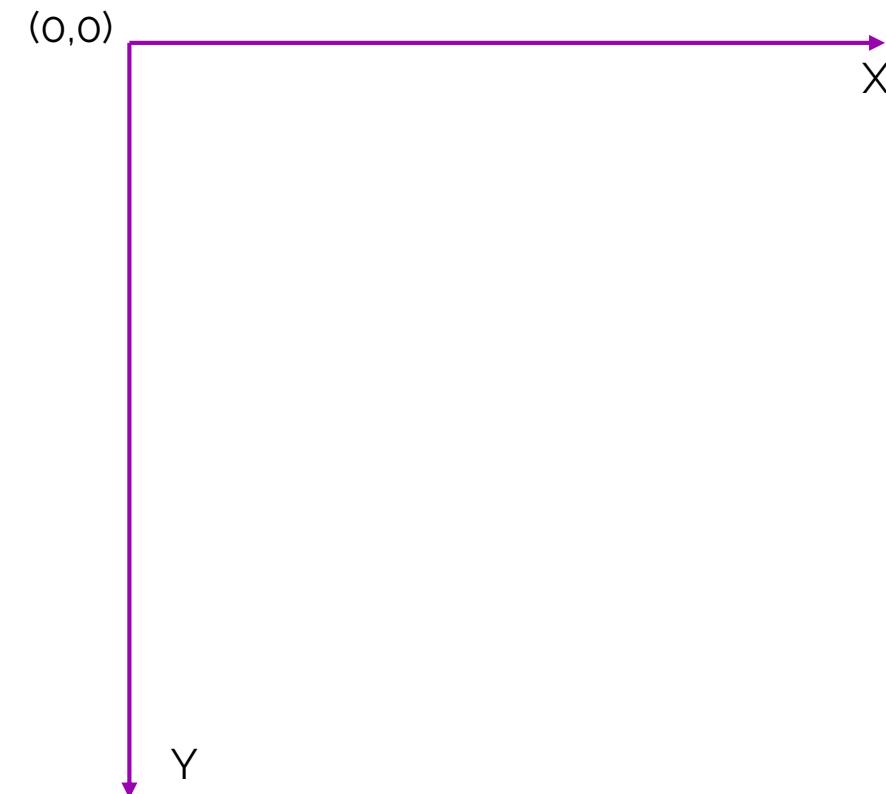
Les styles sont des values, on doit les définir au même endroit que les chaînes de caractères.

Voici la forme standard d'un style :

```
<resources>
  <style name="nom_du_style" parent="nom_du_parent">
    <item name="propriete_1">valeur_de_la_propriete_1</item>
    <item name="propriete_2">valeur_de_la_propriete_2</item>
    <item name="propriete_3">valeur_de_la_propriete_3</item>
    ...
    <item name="propriete_n">valeur_de_la_propriete_n</item>
  </style>
</resources>
```

# Le système de coordonnées

Le système de coordonnées des interfaces graphiques est généralement inversé par rapport à ce que l'on connaît des mathématiques, il s'adapte en effet au sens de lecture, il va de gauche à droite et de haut en bas.



# Le système de coordonnées



Voici quelques **informations utiles** :

- Sur l'axe X, plus on se déplace vers la droite, plus on s'éloigne de 0.
- Sur l'axe Y, plus on se déplace vers le bas, plus on s'éloigne de 0.
- Pour exprimer une coordonnée, on utilise la notation (X, Y).
- L'unité est le **pixel**.
- Le point en haut à gauche a pour coordonnées (0, 0).
- Le point en bas à droite a pour coordonnées (largeur de l'écran, hauteur de l'écran).

# La classe view



Les **layouts** et les **vues** sont les éléments qui permettent de réaliser les interfaces graphiques :

- Les vues sont des éléments qui composent les interfaces graphiques.
- Les layouts sont des modèles qui permettent de disposer les vues.

Les vues sont des éléments qui héritent de la classe **view**.

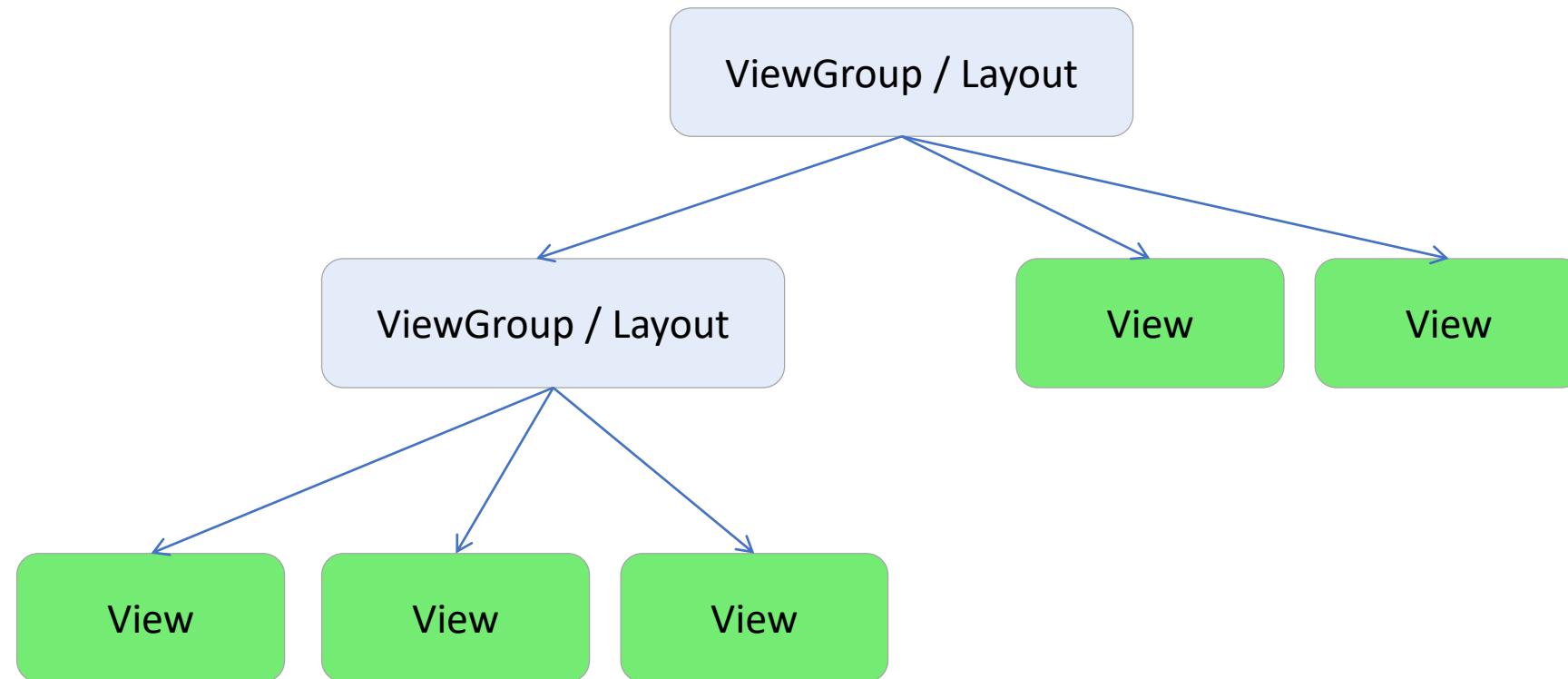
Les vues englobent deux notions :

- Les Layouts, qui permettent de mettre en page une interface ;
- Les Widgets, qui sont les éléments mis en forme via les layouts.

Cette classe contient les notions de disposition d'écran, interaction, etc.

# ViewGroup

Une ou plusieurs vues peuvent être regroupées dans ce que l'on appelle "ViewGroup"



# Les vues



Les vues sont enregistrées en XML :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.axopen.tp2.MainActivity">

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello World!" />
</RelativeLayout>
```

# Les vues



layout:width	match_parent
layout:height	match_parent
▶ layout:margin	[?, ?, 33dp, ?, ?, ?]
layout:alignEnd	<input checked="" type="checkbox"/>
layout:alignParentEnd	<input type="checkbox"/>
layout:alignParentStart	<input type="checkbox"/>
layout:alignStart	
layout:toEndOf	
layout:toStartOf	
▶ layout:alignComponent	[top:bottom]
▶ layout:alignParent	[right]
layout:centerInParent	
<b>style</b>	
accessibilityLiveRegion	
accessibilityTraversalAfter	
accessibilityTraversalBefore	
alpha	
<b>background</b>	
backgroundTint	
backgroundTintMode	
<b>baselineAligned</b>	<input type="checkbox"/>
baselineAlignedChildIndex	
clickable	<input type="checkbox"/>
<b>collapseColumns</b>	

Android Studio permet de gérer les propriétés (hauteur, largeur, gravité, etc) des vues dans la fenêtre dédiée :

Layouts et Widgets partagent des **attributs communs** :

- Layout\_width : Largeur que prend la vue (la place sur l'axe horizontal).
- Layout\_height : Hauteur que prend la vue (la place sur l'axe vertical).

Ces **deux attributs** peuvent avoir les valeurs suivantes :

- match\_parent : la vue prendra autant de place que son parent sur l'axe concerné.
- wrap\_content : la vue prendra autant de place que ses enfants sur l'axe concerné.
- Une valeur numérique précise avec une unité :
  - dp ou dip : il s'agit d'une unité qui est indépendante de la résolution de l'écran.
  - sp : même principe que dp, sauf qu'elle est adaptée pour définir la taille d'une police de caractères.

# Identification des vues



Via l'attribut « android :id », on va définir un id unique pour chaque vue.

Il est nécessaire de déclarer un identifiant pour chacune des vues créées afin de pouvoir y accéder.

## On utilisera la syntaxe :

- @+C/I pour générer un id automatiquement (privilégier).
- @C/I pour gérer les id dans un fichier de ressource particulier.

## Remarque:

- C représente la classe dans laquelle l'identifiant sera créé. Par défaut, C vaut id.
- I est le nom de l'identifiant, qui doit être unique au sein de la classe.

# Interaction avec les vues



On peut interagir avec une vue via son identifiant.

On utilise alors la méthode public **View.findViewById (int id)**, qui retourne une vue (attention donc à bien la caster).

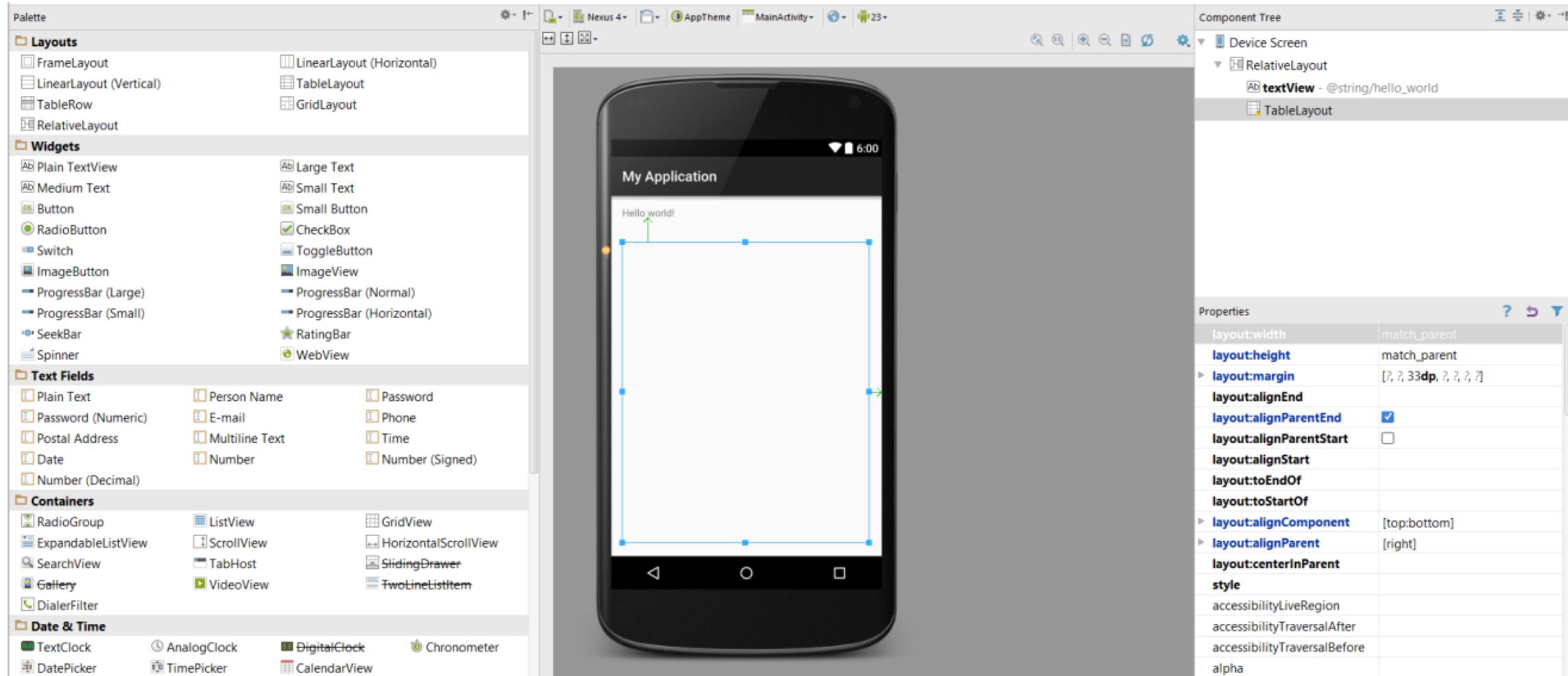
Cet exemple de code montre comment instancier un bouton nommé « bouton1 » :

```
//Instanciation du bouton  
Button nouvellePartieBtn = (Button) findViewById(R.id.button1);
```

# Utiliser l'IDE pour créer des vues



Il est possible de créer des vues via l'IDE:



# Widget



Les widgets sont les éléments qui vont permettre les interactions avec les IHM, ainsi que d'afficher du contenu.

On y retrouve les éléments tels que :

- **Button** : un bouton cliquable.
- **CheckBox** : une case à cocher.
- **TextView** : affiche une chaîne de caractères non modifiable.
- **EditText** : un champ de texte éditable.
- **DatePicker** : Sélection de dates.
- **RadioButton** : Représente les boutons radios.
- **Toast** : Un pop-up message qui s'affiche sur l'écran.
- **ImageButton** : Une image qui se comporte comme un bouton.

# Événements sur les widgets



La programmation sous Android est dite événementielle.

Pour capturer ces événements, on utilisera des **listeners**. Pour chaque événement que l'on souhaitera capturer, on définira un **listener** sur la vue concernée.

La syntaxe sera alors du type :

- View.OnClickListener : cette méthode sera appelée lorsque l'utilisateur appuiera sur un widget.
- View.OnLongClickListener : cette méthode sera appelée lorsque l'utilisateur appuiera longtemps sur un widget.
- View.OnKeyListener : cette méthode sera appelée lors de l'appui sur une touche spécifique.

# Listener



La méthode la plus simple pour implémenter un listener consiste à définir un listener sur une vue :

```
Button nouvellePartieBtn = (Button) findViewById(R.id.button1);
```

On peut alors définir les méthodes associées:

```
public void onClick(View v) {
    Intent intent = new Intent(MainActivity.this, Partie.class);
    startActivity(intent);
}
```

# Le menu



Chaque activité est capable d'avoir son menu propre. La meilleure façon est de le créer en XML.

La racine de ce menu est de type <menu>/

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <!-- Code -->
</menu>
```

Ce menu doit être peuplé avec des <item> qui peuvent être personnalisés à l'aide de plusieurs attributs:

- **android:id** : permet d'identifier de manière unique un <item>.
- **android:icon**, pour agrémenter votre <item> d'une icône.
- **android:title**, qui sera son texte dans le menu.
- on peut désactiver par défaut un <item> avec l'attribut android:enabled="false".

# Le menu



## Exemple:

```
<item>
<menu>
<item />
<!-- d'autres items-->
</menu>
</item>
```

Le sous-menu s'ouvrira alors dans une nouvelle fenêtre.

Lorsqu'on utilise un menu, il faut parcourir le fichier XML de celui-ci. Pour cela, on va surcharger la méthode : **boolean onCreateOptionsMenu (Menu menu)** d'une activité. Cette méthode est lancée au moment de la première pression du bouton qui fait émerger le menu.

# Le menu



Si vous souhaitez que le menu évolue à chaque pression du bouton, alors il vous faudra surcharger la méthode **boolean onPrepareOptionsMenu (Menu menu)**.

Pour parcourir le XML, on va le désérialiser.

Exemple de code type dès qu'on a constitué un menu en XML :

```
@Override  
public boolean onCreateOptionsMenu(Menu menu) {  
    super.onCreateOptionsMenu(menu);  
    MenuInflater inflater = getMenuInflater();  
    //R.menu.menu est l'id de notre menu  
    inflater.inflate(R.menu.menu, menu);  
    return true;  
}
```

# Le menu dynamique

Sur un menu on peut, avec la méthode **MenuItem add (int groupId, int objectId, int ordre, CharSequence titre)**, construire un menu de manière programmatique.

De manière identique et avec les mêmes paramètres, **MenuItem addSubMenu (int groupId, int objectId, int ordre, CharSequence titre)**, permet de construire un sous-menu.



# Layout et Composants

" Cycle Developpeur "

# Les Layout



Les layouts peuvent être vu comme des groupes de composants. Il sont eux même des composants, de ce fait l'imbrication de ceux-ci est possible. Nous allons voir ici les layout :

- LinearLayout,
- RelativeLayout,
- TableLayout,
- GridLayout etc.

# Les propriétés communes

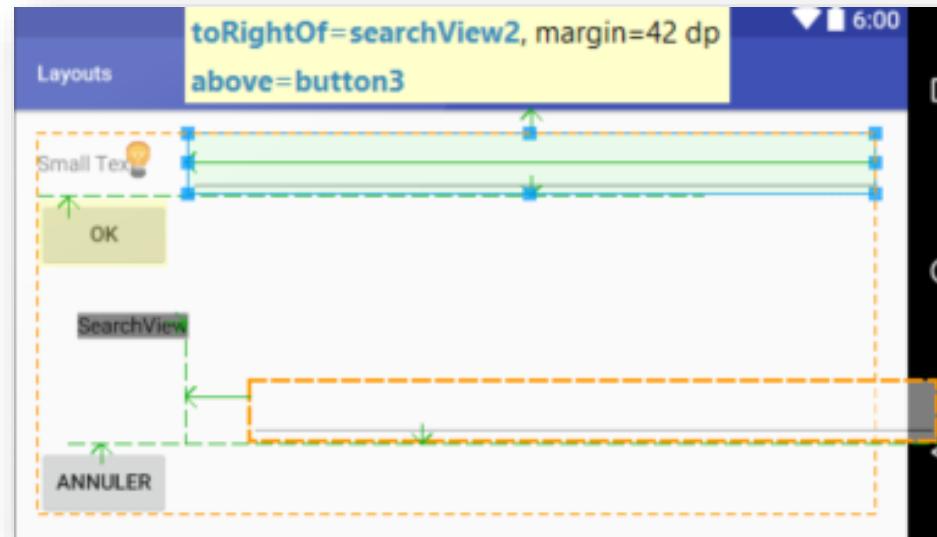


Les composants et les layout possèdent des propriétés communes comme :

- la largeur (width)
- la hauteur (height)
- alignement du parent (align\_parent)
- visibility
- marges externes (margin)
- marges internes (padding)
- etc.

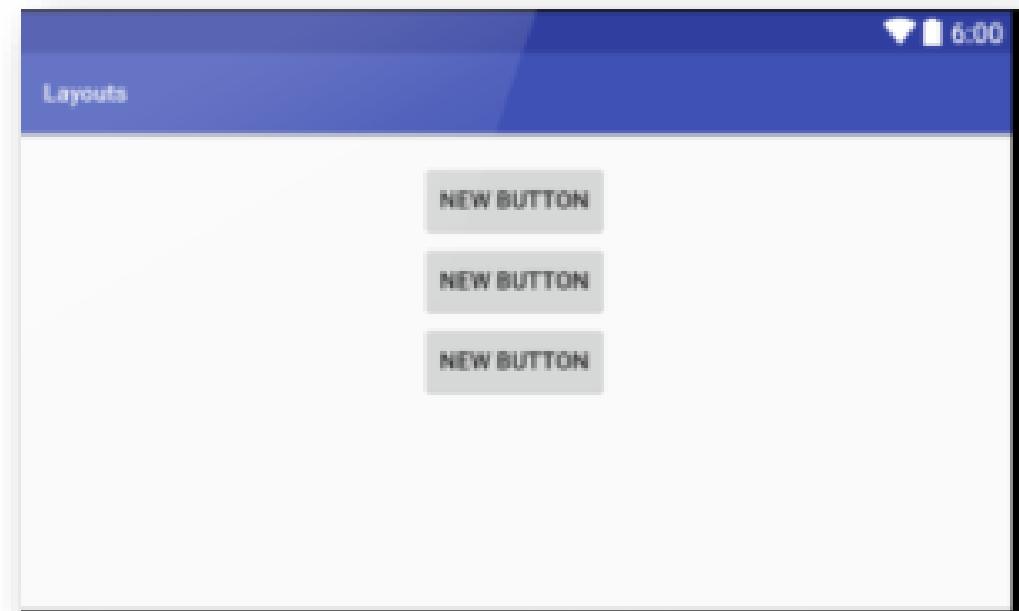
# Le RelativeLayout

Ce layout permet de positionner un composant en relation avec les autres composants de l'application.



# Le LinearLayout

Le linear layout est l'un des layouts le plus utile sous android il permet de positionner des éléments verticalement à la suite. Il introduit entre autre la propriété d'orientation. On peut donc définir ce layout horizontalement ou verticalement.



# Le TableLayout

C'est la représentation des composants dans un format tabulaire.

Son apparence simple est trompeuse. Il introduit notamment des propriétés de span, et se compose de TableRow.



# GridLayout



L'un des gestionnaires de mise en page les plus complexes. Il dispose les composant dans un tableau à deux dimensions.

A la différence du TableLayout, il ne possède pas de TableRow, et possède des spans sur les lignes et sur les colonnes

# Les composants



Cette section vise à présenter des composants utiles pour la création des interfaces :

- Le Button
- Le Spinner
- Les CheckBox
- Les RadioButton et RadioGroup
- Le composant de tabulation
- Le ListView
- Le GridView
- La progressBar
- etc.

# Composant : Button

Le **Button** est l'un des composants les plus utilisé et basique d'une interface graphique.  
Il déclenche un événement au click

## Attributs utiles

- onClick :** précise l'action à effectuer au click
- text :** précise le texte à afficher

SE CONNECTER

# Composant : EditText

Le **EditText** est un composant pour la saisie du texte.

## Attributs utiles

**inputType:** précise la méthode pour la saisie de la donnée. (*Entier, Autocomplétion etc.*)

**text :** précise le texte à afficher



# Composant : TextView



Le **TextView** est un composant pour l'affichage du texte.

Small Text

Medium Text

Large Text

## Attributs utiles

**text** :      *précise le texte à afficher*

# Composant : ToggleButton

Le **ToggleButton** est un bouton pressoir à deux état On et Off.

Le texte est bien sur personnalisable

## Attributs utiles

**textOn** : précise le texte affiché pour l'état On

**textOff** :           précise le texte affiché pour l'état Off

**checked** :           coche ou décoche un bouton



# Composant : RadioButton

Les boutons Radio (**RadioButton**) doivent se trouver dans un **Radiogroup**.

Le choix doit être unique dans le groupe.

## Attributs utiles

**checked :** coche ou décoche le bouton radio

- 
- New RadioButton
  - New RadioButton
  - New RadioButton

# Composant : CheckBox

Les **checkbox** sont des cases à cocher.

Possédant donc deux états :

- Coché
- Décoché

## Attributs utiles

**checked** : coche ou décoche le bouton radio



# Composant : ProgressBar

La **progressBar** est un composant pour faire patienter l'utilisateur, il peut être représenté sous la forme d'un « sablier » d'attente (sans connaissance de la progression) ou sous la forme d'une barre de progression.



# Composant : ListView

Le composant **ListView** est un composant puissant pour l'affichage d'une liste d'élément. Il sépare la couche données et affichage en proposant la notion d'adapter, adaptant des templates de conceptions aux données

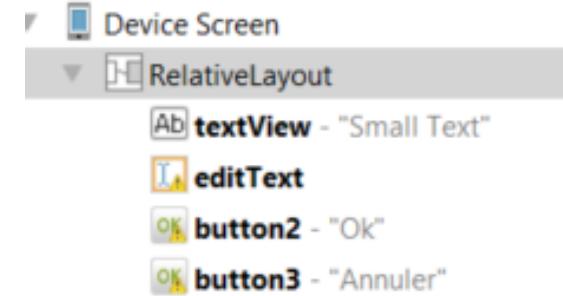
## Méthode utiles

**setAdapter:** *détermine l'adaptateur pour l'affichage graphique*

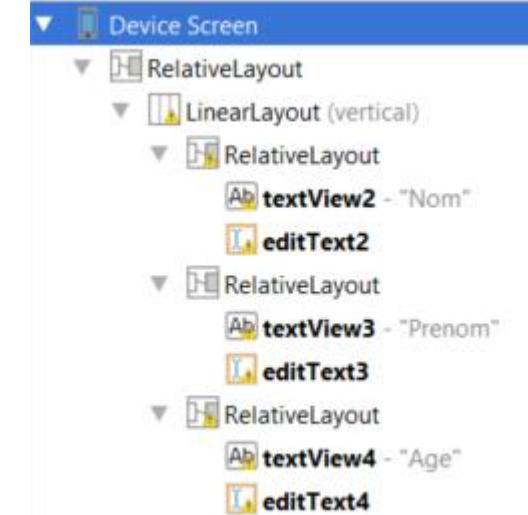
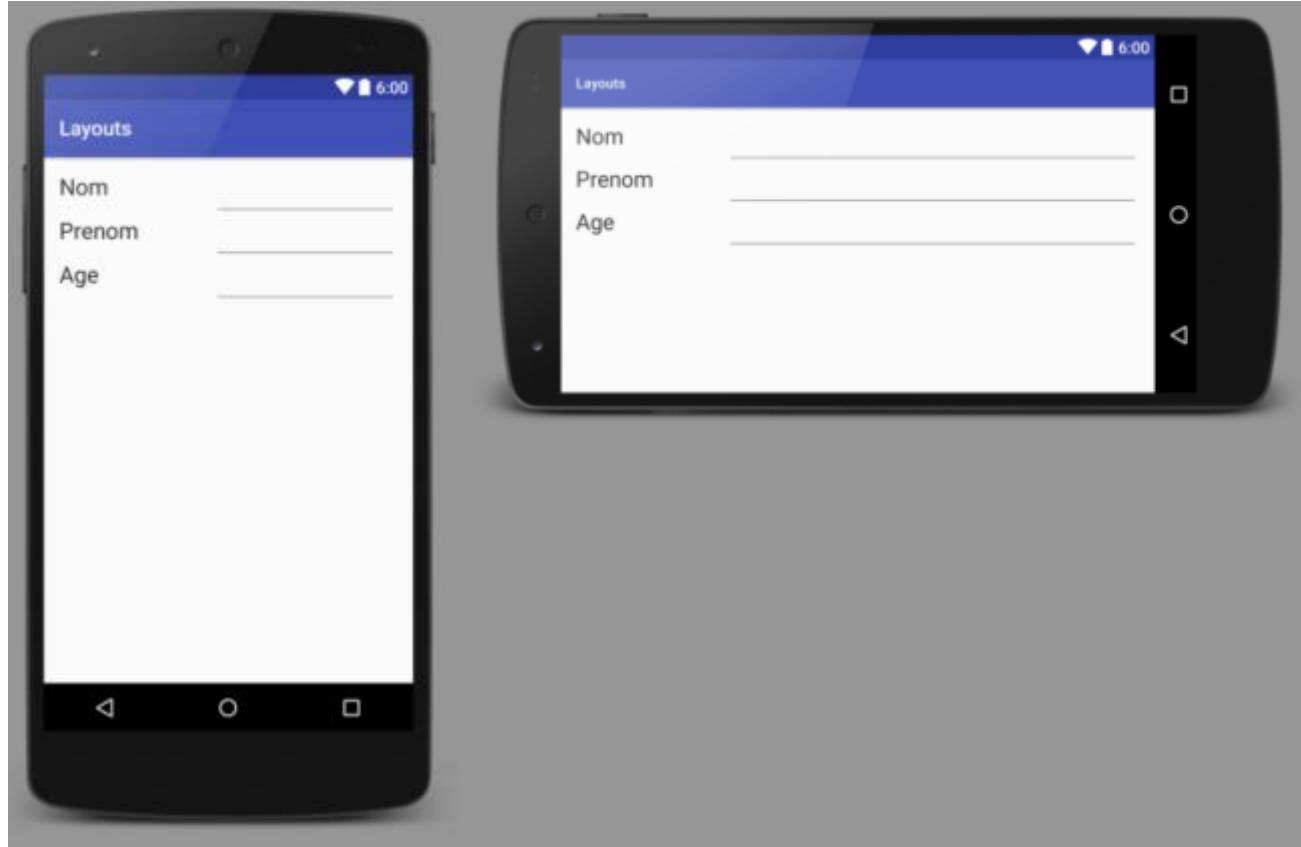


	Tranche de porc	7 €
	Crevette Ananas	17 €
	Panacotta	4 €
	Emincé de Porc	9 €
	Crevette Exotiques	17 €

# Deviner le Layout (2)



# Deviner le Layout (1)





# Le Modèle de Composants

" Cycle Developpeur "

# Activité



Une activité est la composante principale pour une application Android. Elle représente l'implémentation et les interactions de vos interfaces.

**Exemple** : si une application liste les fichiers présents sur un périphérique, le projet sera décomposé comme ci-dessous :

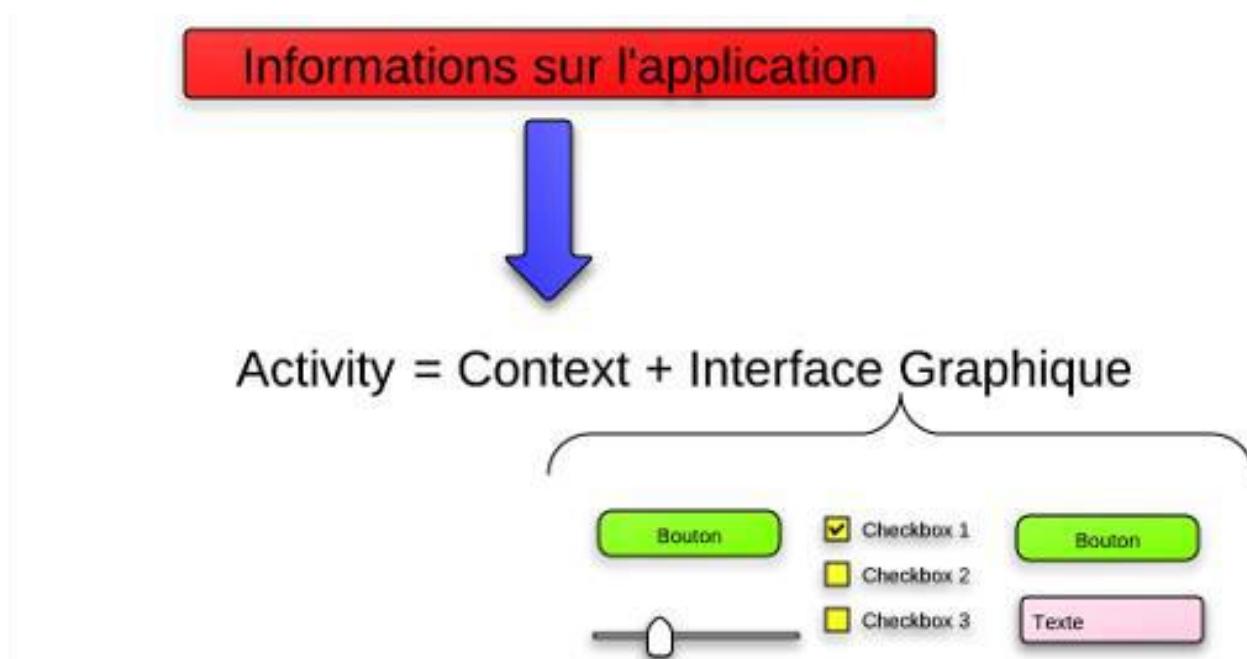
- Une vue pour afficher la liste des fichiers.
- Une activité pour gérer le remplissage et l'affichage de la liste.

**Remarque** : chaque page n'est reliée qu'à une activité. On peut donc résumer l'usage des activités à une simple formule :

**Une page = Une activité.**

# Activité

Une activité contient des informations sur l'état actuel de l'application : ces informations s'appellent le context.



Les activités **héritent de la classe Activity**. La classe Activity **hérite de l'interface Context** dont le but est de représenter tous les composants d'une application

# Etat d'une activité



Android est un système multi tâche ; néanmoins, il est parfois judicieux que le système switch entre plusieurs applications.

*Exemple :* Lorsque le téléphone sonne, fermer une vidéo et ouvrir la fenêtre pour répondre à un appel.

Pour pouvoir réaliser ce mécanisme de priorisation, il est nécessaire de mettre en place 2 concepts :

- Prioriser les applications. Ainsi, en cas de concurrence, c'est l'application qui aura la priorité la plus élevée qui coupera l'autre.
- Les activités existeront dans plusieurs états au cours de leurs vies, généralement un état actif durant l'utilisation, et un état de pause quand l'utilisateur reçoit un appel.

# Fonctionnement de la pile d'activités



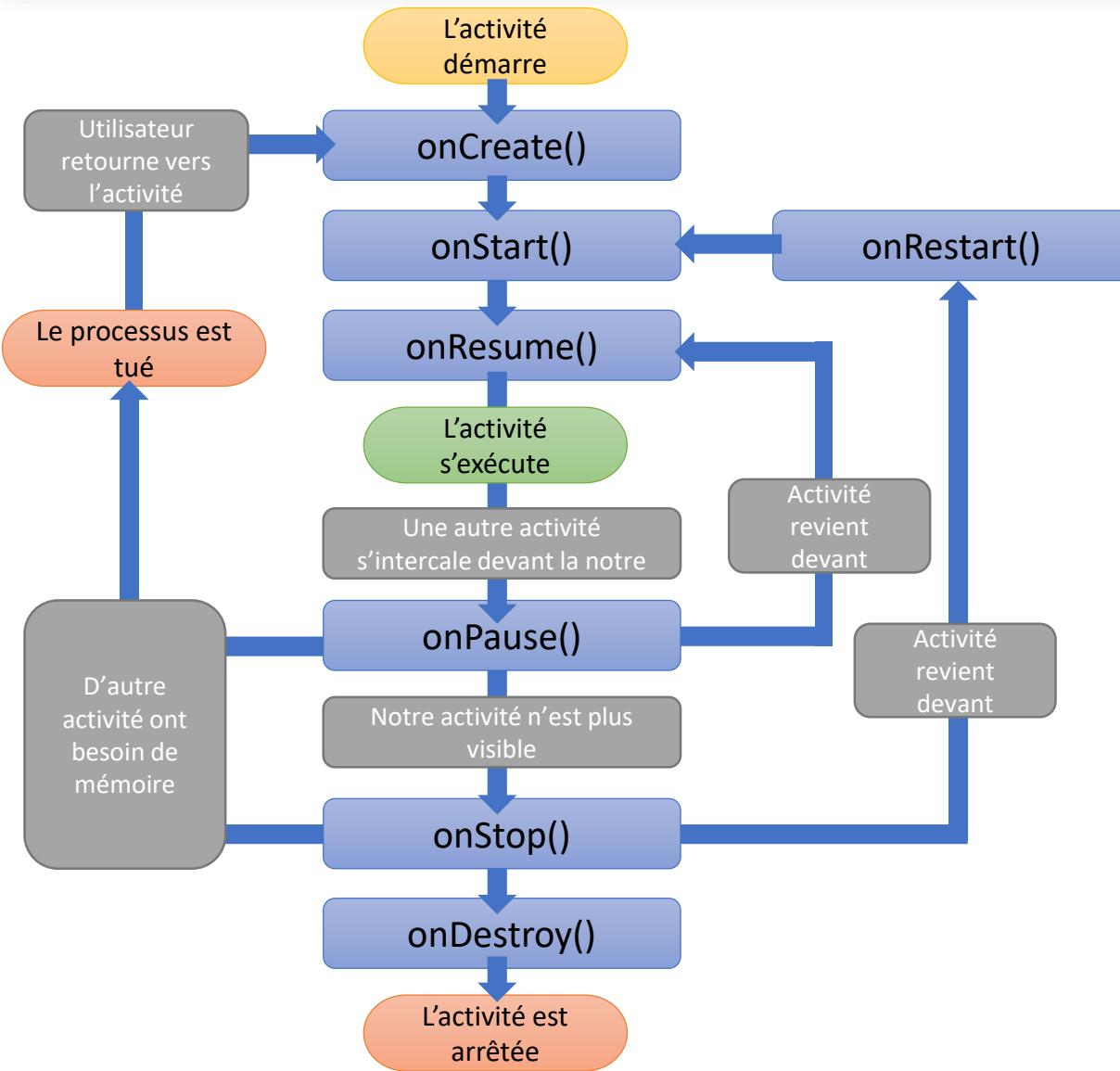
L'activité que voit l'utilisateur est celle qui se trouve au-dessus de la pile.

Lorsqu'un appel arrive, il se place au **sommet** de la pile et c'est lui qui s'affiche à la place de votre application, qui n'est plus qu'à la deuxième place.

Une activité peut se trouver dans trois états qui se différencient surtout par leur visibilité :

- **Active ou running** : L'activité est visible, l'utilisateur agit au moment même.
- **Pause** : L'activité est partiellement visible à l'écran. L'application sur laquelle l'utilisateur agissait n'a plus le focus, c'est l'application au plus au niveau de la pile qui l'a.
- **Arrêté** : L'activité est masquée par une autre activité. L'application n'a plus le focus.

# Cycle de vie



Durant sa vie, une activité passe par **differents états**. L'activité n'a **pas de contrôle** sur son état, et ce sont les interactions avec le système qui la font passer d'un état à un autre.

Ces différents états sont **gérés via des méthodes**.

# Cycle de vie



- **onCreate()** : Appelé lors de la première création de l'activité. C'est ici que l'on doit initialiser les liens entre les données et les vues. Toujours suivi par onStart().
- **onRestart()** : Appelé après votre activité a été arrêté, avant d'être à nouveau lancé. Toujours suivi par onStart().
- **onStart()** : Appelé juste avant que l'activité soit visible par l'utilisateur. Suivi par onResume() si visible au premier plan ou onStop() si l'application est masquée.
- **onResume()** : est lancé lorsque votre application est passée en avant plan. Vous pourrez donc (re)lancer vos threads.

# Cycle de vie



- **onPause()** : Lancée lorsqu'une autre activité va s'afficher à l'avant plan. C'est le moment de sauvegarder toutes les données/informations saisies par l'utilisateur pour l'activité courante. Le système peut décider par la suite de mettre fin à votre activité par exemple à cause d'un manque de mémoire.
- **onStop()** : Lancée lorsque votre activité n'est plus visible. Votre processus va maintenant passer en sommeil.
- **onDestroy()** : Fin au cycle de vie de l'activité. Les ressources seront libérées, les fichiers temporaires sont purgés. onCreate() devra être exécutée pour obtenir à nouveau l'activité.

# Les intents



Créer un **intent** explicite est très simple puisqu'il suffit de donner un **Context** qui appartienne au package où se trouve la classe de destination :

```
Intent intent = new Intent(Activite_de_depart.this, Activite_de_destination.class);
```

On utilise ensuite la fonction **void startActivity (Intent intent)**.

# Les listes et adaptateurs



Si on souhaite afficher des objets d'un type particulier, composés de **plusieurs** informations, plusieurs questions se posent :

- Quelle est **l'information à afficher** pour l'élément?
- Que faire quand **on clique sur un élément** de la liste ?
- Comment sont **représentées** les informations ?

Pour résoudre ces problèmes, on va donc utiliser des **Listes**.

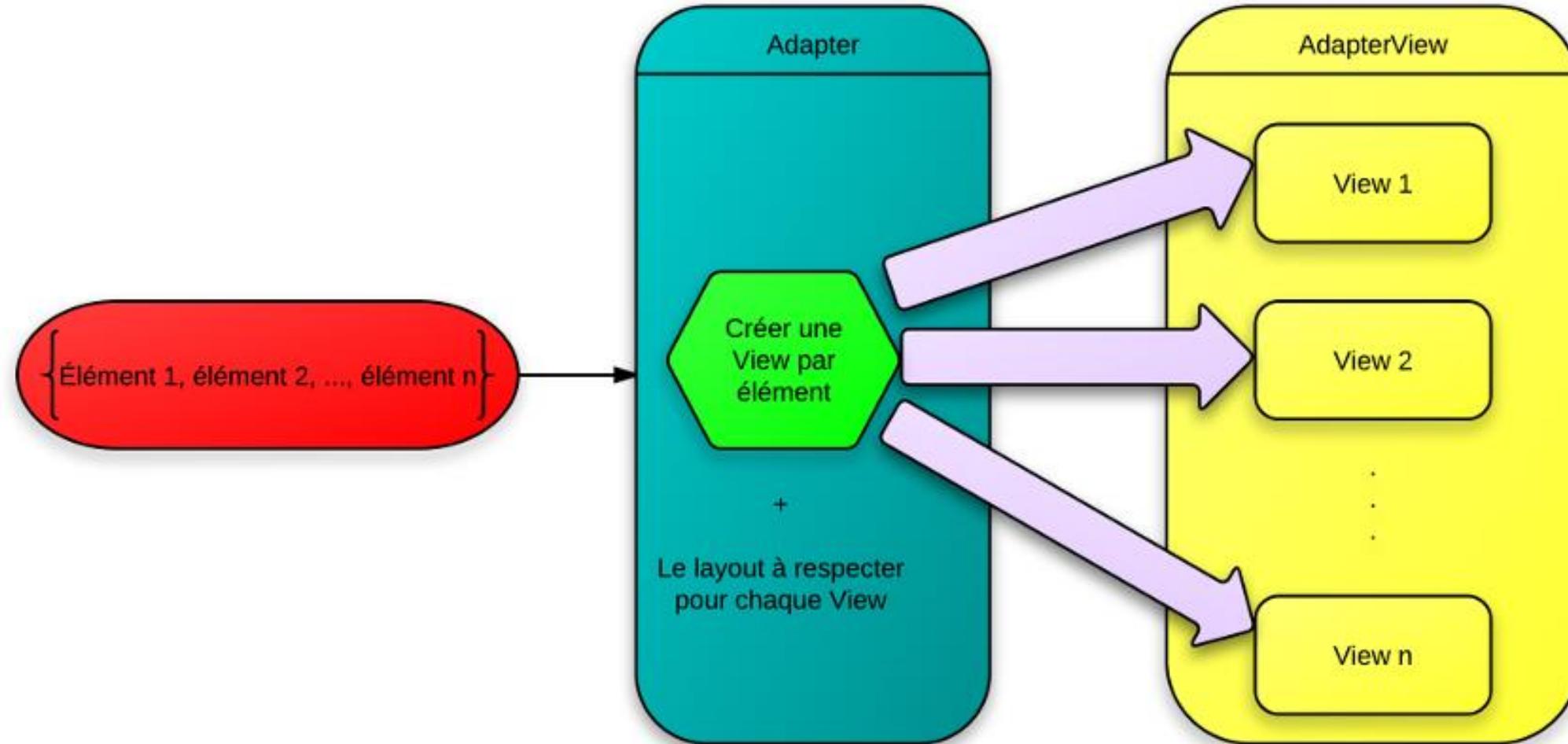
# Les listes et adaptateurs



La gestion des listes se divise en deux parties distinctes :

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

# Les listes et adaptateurs



# Les adaptateurs



La classe **Adapter** est une interface qui définit les comportements généraux des adaptateurs.

Si on veut construire un widget simple, on retiendra trois principaux adaptateurs :

1. **ArrayAdapter**, qui permet d'afficher les informations simples.
2. **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).
3. **CursorAdapter**, pour adapter le contenu qui provient d'une base de données. On y reviendra dès qu'on abordera l'accès à une base de données.

# ArrayAdapter



C'est un adaptateur pour les informations stockées dans un tableau.

**ArrayAdapter** est un adaptateur prenant en paramètres un contexte, un id et un tableau d'objets:

- **Contexte** : le contexte.
- **Id** : référence à un layout. Il déterminera la mise en page de l'élément. On a la possibilité de créer une ressource de layout personnalisée. Android met à **disposition certains layouts**, qui dépendent beaucoup de la liste dans laquelle vont se trouver les widgets.
- **Objects[]** : la liste ou le tableau des éléments à afficher.

# SimpleAdapter

Le SimpleAdapter est utile pour afficher simplement plusieurs informations par élément. Chaque information de l'élément aura une vue dédiée qui affichera l'information voulue.

**SimpleAdapter** est un adaptateur prenant en paramètres un context, un tableau d'objets, un id, un tableau de String et un tableau de int:

- **String[]**: les clés des informations à afficher dans chaque élément.
- **Int[]**: les layout à appliquer à chacun des éléments.

# Méthodes des Adaptateurs

Voici quelques méthodes communes aux adaptateurs :

- **void add (T object)**: ajoute un objet à un adaptateur.
- **void insert (T object, int position)**: insère un objet à une position particulière.
- **T getItem (int position)**: récupère un objet dont on connaît la position.
- **int getPosition (T object)**: récupère la position d'un objet précis.
- **void remove (T object)**: supprime un objet.
- **void clear()**: vide complètement l'adaptateur.

Par défaut, un **ArrayAdapter** affichera pour chaque objet de la liste le résultat de la méthode **String toString()** associée et l'insérera dans une **TextView**.

# AdapterView



## L'AdapterView :

- Lie les sous-éléments créés via l'adaptateur et les affiche sous forme de liste.
- Gère les interactions avec les utilisateurs : l'adaptateur s'occupe des éléments en tant que données, alors que l'AdapterView s'occupe de les afficher et veille aux interactions avec un utilisateur.

## Les 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 adaptateur à une AdapterView, on utilise la méthode **void setAdapter (Adapter adapter)**, qui se chargera de peupler la vue.

# ListView



Une **ListView** est une liste de vues comprenant des items ; pour afficher une liste d'items dans celle-ci, il lui faut un adaptateur de données.

**Exemple :**

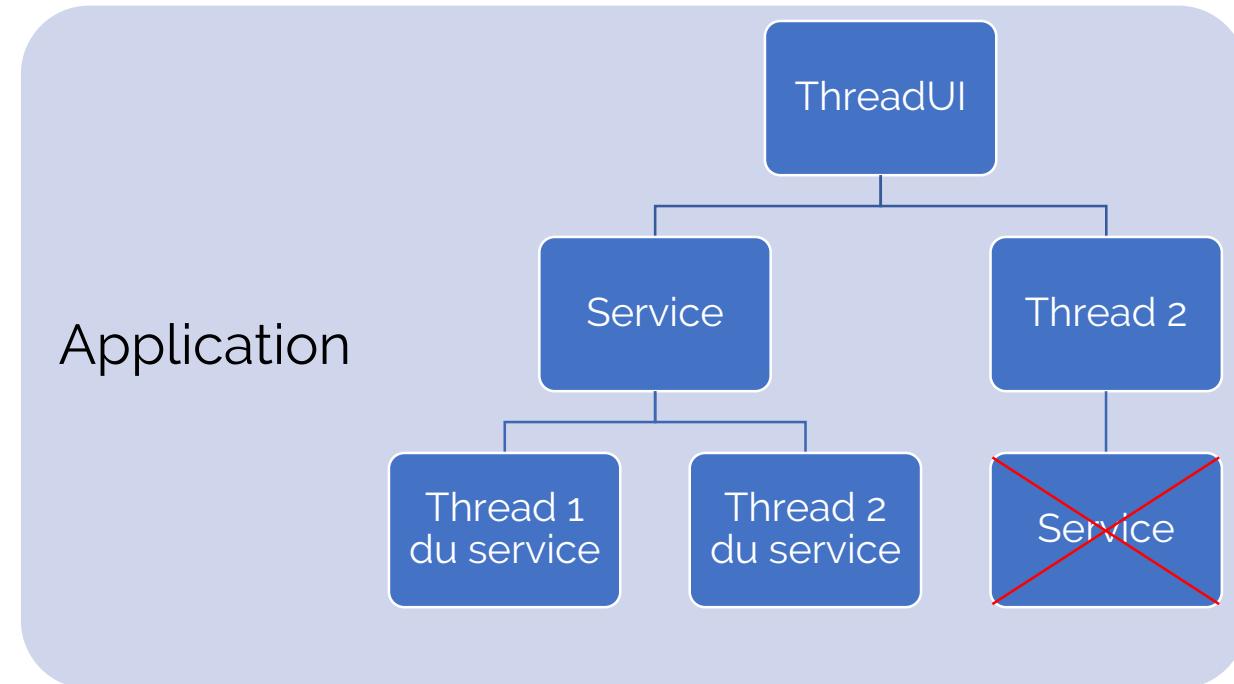
```
public class TutoListesActivity extends Activity {  
    ListView liste = null;  
  
    @Override  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.main);  
  
        liste = (ListView) findViewById(R.id.listView);  
        List<String> exemple = new ArrayList<String>();  
        exemple.add("Item 1");  
        exemple.add("Item 2");  
        exemple.add("Item 3");  
  
        ArrayAdapter<String> adapter = new ArrayAdapter<String>(this, android.R.layout.simple_list_item_1, exemple);  
        liste.setAdapter(adapter);  
    }  
}
```

# Qu'est ce que c'est ?

Un service se lance par défaut dans le même processus que celui de l'appelant.

Les services se lancent dans le ThreadUI. Ils ne sont pas conçus pour être exécutés en dehors de ce thread.

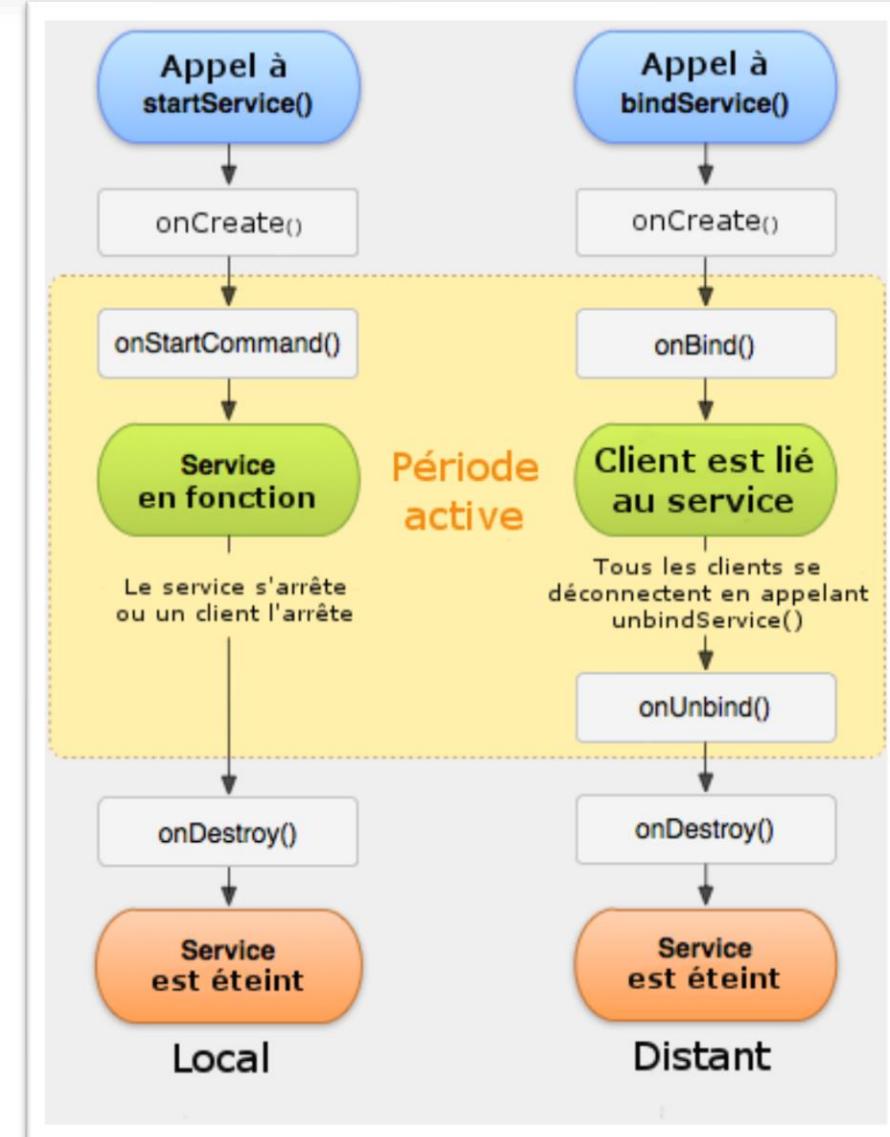
En revanche, on peut lancer un thread dans le service si les opérations menées dans le service risquent d'affecter l'interface graphique.



# Cycle de vie

De manière analogue aux activités, les services traversent plusieurs phases pendant leur vie.

La transition entre ces phases est matérialisée par des méthodes de callback. Le cycle des services est plus facile à maîtriser que celui des activités car il possède moins de phases.



# Les services locaux



Il sont appelés à partir d'une activité avec la méthode startService(Intent service).

La variable renvoyée donne le package accompagné du nom du composant qui vient d'être lancé.

La création d'un service est symbolisée par la méthode « void onCreate() ».

La méthode appelée ensuite est « int onStartCommand(Intent intent, int flags, int startId) ».

Si le service existait déjà, alors « onStartCommand() » est directement appelée sans passer par « onCreate() ».

- Intent, lance le service.
- flags, la nature de l'intent.
- startId, pour l'indentification. startId vaut 1 s'il s'agit du premier lancement, puis 2 pour le deuxième, etc.

Cette méthode retourne un entier. Cet entier doit en fait être une constante qui détermine le comportement du système s'il est tué.

# Les flags



## **START\_NOT\_STICKY**

Si le système tue le service, alors ce dernier ne sera pas recréé. Il faudra donc effectuer un nouvel appel à startService() pour relancer le service.

## **START\_STICKY**

Si le système doit tuer le service, alors il sera recréé mais sans lui fournir le dernier Intent qui l'avait lancé. Ainsi, le paramètre intent vaudra null.

## **START\_REDELIVER\_INTENT**

Si le système tue le service, il sera recréé et dans onStartCommand(), le paramètre intent sera identique au dernier intent qui a été fourni au service. START\_REDELIVER\_INTENT est indispensable si vous voulez être certains qu'un service effectuera un travail complètement.

# Le paramètre flags



Le paramètre flags de onStartCommand() montre la nature de l'intent qui a lancé le service :

- 0 si il n'y a rien de spécial à dire.
- START\_FLAG\_REDELIVERY si l'intent avait déjà été délivré et qu'il l'est à nouveau parce que le service avait été interrompu.
- Enfin, vous trouverez aussi START\_FLAG\_RETRY si le service redémarre alors qu'il s'était terminé de manière anormale.

# Priorité



Une fois sorti de la méthode `onStartCommand()`, le service est lancé. Un service continuera à fonctionner jusqu'à ce que vous l'arrêtiez ou qu'Android le fasse de lui-même pour libérer de la mémoire RAM, comme pour les activités.

Les services sont plus susceptibles d'être détruits qu'une activité située au premier plan, mais plus prioritaires que les autres processus qui ne sont pas visibles.

La priorité diminue avec le temps : plus un service est lancé depuis longtemps, plus il a de risques d'être détruit.

Pour arrêter un service, il est possible d'utiliser `void stopSelf()` depuis le service ou `boolean stopService(Intent service)` depuis une activité, auquel cas il faut fournir `service` qui décrit le service à arrêter.

# Les services distants



Comme les deux types de services sont assez similaires, seules les différences sont spécifiées.

On utilisera cette fois `boolean bindService(Intent service, ServiceConnection conn, int flags)` afin d'assurer une connexion persistante avec le service.

Un ServiceConnection est une interface pour surveiller l'exécution du service distant. Il existe deux méthodes que vous devrez redéfinir :

1. `void onServiceConnected(ComponentName name, IBinder service)` qui est appelée quand la connexion au service est établie, avec un IBinder qui correspond à un canal de connexion avec le service.
1. `void onServiceDisconnected(ComponentName name)` qui est appelée quand la connexion au service est perdue, en général, parce que le processus qui accueille le service a planté ou a été tué.

# IBinder

Un IBinder est un pont entre un service et une activité, mais au niveau du service.

Les IBinder permettent au client de demander des choses au service. Pour créer cette interface, il faut savoir que le IBinder qui sera donné à onServiceConnected(ComponentName, IBinder) soit envoyé par la méthode IBinder onBind(Intent intent) dans Service.

Pour créer un IBinder. La méthode la plus simple, consiste à permettre à l'IBinder de renvoyer directement le Service de manière à pouvoir effectuer des commandes dessus.

```
public class MainService extends Service {  
  
    @Override  
    public IBinder onBind(Intent intent) { return mBinder; }  
  
    // Attribut de type IBinder  
    private final IBinder mBinder = new MonBinder();  
    // Le Binder est représenté par une classe interne  
    public class MonBinder extends Binder {  
        // Le Binder possède une méthode pour renvoyer le Service  
        MainService getService() { return MainService.this; }  
    }  
}
```

# Les Fragments (Principe)



Un fragment peut être vu comme une sous-activité embarquable dans une activité principale.

Le fragment est donc un « composant-activité » il possède aussi un cycle de vie.

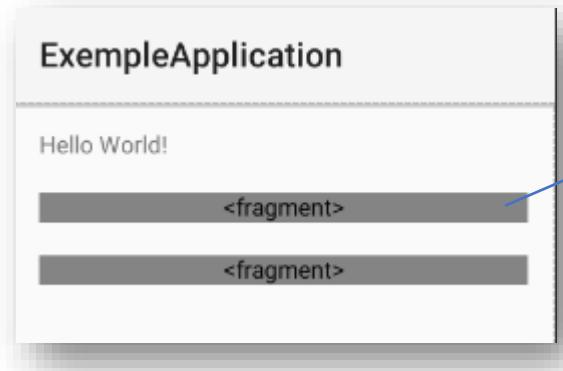
Les écrans de préférences sont des fragments.

# Les Fragments (Déclaration)



Pour créer un Fragment :

```
public class ArticleFragment extends Fragment {  
    @Override  
    public View onCreateView(LayoutInflater inflater, ViewGroup container,  
                             Bundle savedInstanceState) {  
        // Inflate the layout for this fragment  
        return inflater.inflate(R.layout.fragment_article, container, false);  
    }  
}
```



Il faut associer la classe du fragment au composant.

# Les Fragments (Instanciation)



Il est aussi possible d'instancier et d'associer le Fragment au Composant dynamiquement.

Nous passons alors par le FragmentManager :

```
ArticleFragment fragment = new ArticleFragment();  
  
getSupportFragmentManager().beginTransaction()  
    .add(R.id.fragment2, fragment).commit();
```

# Les Fragments (Gestion)



Les FragmentManager va nous permettre de :

- Supprimer des fragments existant
- Remplacer des fragments par d'autres
- Ajouter des fragments
- Faire communiquer les Fragments entre eux

Exemple de code pour le remplacement de Fragments :

```
getSupportFragmentManager()
    .beginTransaction()
    .replace(R.id.fragment, new ArticleFragment2())
    .addToBackStack(null)
    .commit();
```

# Les Fragments (Gestion)



Exemple de code pour la suppression de Fragments :

```
getSupportFragmentManager()
    .beginTransaction()
    .remove(fragment)
    .addToBackStack(null)
    .commit();
```



# La Persistance des Données

" Cycle Developpeur "

# Différentes persistances



Android fournit plusieurs méthodes pour faire persister les données applicatives:

- La persistance des **données de l'activité**.
- Un mécanisme de **sauvegarde clé/valeur**, utilisé pour les fichiers de préférences (appelé SharedPreferences).
- Des entrées/sorties de type **fichier**.
- Une base de données basée sur **SQLite**.

La persistance des données des activités est gérée par l'objet **Bundle** qui permet de restaurer les View possédant un id.

# Représentation XML



Exemple de déclarations de préférences XML, à stocker dans res/xml/preferences.xml:

```
<PreferenceScreen xmlns:android="http://schemas.android.com/apk/res/android"
    android:key="first_preferencescreen">
<CheckBoxPreference
    android:key="wifi_enabled"
    android:title="WiFi" />
<PreferenceScreen
    android:key="second_preferencescreen"
    android:title="WiFi settings">
    <CheckBoxPreference android:key="prefer_wifi" android:title="Prefer WiFi" />
</PreferenceScreen>
</PreferenceScreen>
```

# SharedPreferences



La classe **SharedPreferences** permet de gérer des paires de clé/valeurs associées à une activité.

On récupère un tel objet par l'appel à **getPreferences**:

```
SharedPreferences prefs = getPreferences(Context.MODE_PRIVATE);
String nom = prefs.getString("login", null);
Long nom = prefs.getLong("taille", null);
```

La méthode `getPreferences(int)` appelle en fait `getPreferences(String, int)` à partir du nom de la classe de l'activité courante.

- **MODE\_PRIVATE** restreint l'accès au fichier créé à l'application.
- **MODE\_WORLD\_READABLE** et **MODE\_WORLD\_WRITABLE** permettent aux autres applications de lire/écrire ce fichier.

# Activité d'édition de préférences



Pour afficher l'écran d'édition des préférences correspondant à sa description XML, il faut créer une nouvelle activité qui hérite de **PreferenceActivity** et simplement appeler la méthode **addPreferencesFromResource** en donnant l'id de la description XML:

```
public class MyPrefs extends PreferenceActivity {  
    public void onCreate(Bundle savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        addPreferencesFromResource(R.xml.preferences);  
    }  
}
```

# Attributs des préférences



Les attributs suivants sont utiles:

- **android:title**: le string apparaissant comme nom de la préférence
- **android:summary**: Une phrase permettant d'expliciter la préférence
- **android:key**: La clef pour l'enregistrement de la préférence

Pour accéder aux valeurs des préférences, on utilise la méthode **getSharedPreferences** sur la classe **PreferenceManager**. C'est la clef spécifiée par l'attribut **android:key** qui est utilisée pour récupérer la valeur choisie par l'utilisateur.

```
SharedPreferences prefs = PreferenceManager.getSharedPreferences(context);
String login = prefs.getString("login", "");
```

# Attributs des préférences



Des attributs spécifiques à certains types de préférences peuvent être utilisés:

- **android:summaryOn** pour les cases à cocher qui donne la chaîne à afficher lorsque la préférence est cochée.
- **android:dependency** pour faire dépendre une préférence d'une autre.

**Exemple:**

```
<CheckBoxPreference android:key="wifi" ... />
<EditTextPreference android:dependency="wifi" ... />
```

# Attributs des préférences



Une case à cocher se fait à l'aide de CheckBoxPreference:

```
<CheckBoxPreference android:key="wifi"
    android:title="Utiliser le wifi"
    android:summary="Synchronise l'application via le wifi."
    android:summaryOn="L'application se synchronise via le wifi."
    android:summaryOff="L'application ne se synchronise pas."
/>
```

Un champ texte est saisi via EditTextPreference:

```
<EditTextPreference
    android:key="login"
    android:title="Login utilisateur"
    android:summary="Renseigner son login d'authentification."
    android:dialogTitle="Veuillez saisir votre login"
/>
```

# Les fichiers



Android fournit aussi un accès classique au système de fichier.

Pour la gestion des permissions, on retrouve comme pour les préférences les constantes **MODE\_PRIVATE** et **MODE\_WORLD\_READABLE/WRITABLE** à passer en paramètre de l'ouverture du fichier.

En utilisant ces constantes comme un masque, on peut ajouter **MODE\_APPEND** pour ajouter des données.

```
try {
    FileOutputStream out = openFileOutputStream("fichier", MODE_PRIVATE);
    ...
} catch (FileNotFoundException e) {
    ...
}
```

# Les fichiers



Les ressources permettent aussi de récupérer un fichier embarqué dans l'application:

```
Resources res = getResources();  
InputStream is = res.openRawResource(R.raw.fichier);
```

Les fichiers sont à éviter. Mieux vaut alléger l'application au maximum et prévoir le téléchargement de ressources nécessaires et l'utilisation de fournisseurs de contenu.

Android fournit plusieurs parsers XML (Pull parser, Document parser, Push parser).

SAX et DOM sont disponibles.

Une librairie équivalente est aussi disponible: XmlPullParser.

## Exemple :

```
String s = new String("<plop><blup attr=\"45\">Hellow World</blup></plop>");  
InputStream f = new ByteArrayInputStream(s.getBytes());  
XmlPullParser parser = Xml.newPullParser();  
try {  
    // auto-detect the encoding from the stream  
    parser.setInput(f, null);  
    parser.next();  
    Toast.makeText(this, parser.getName(), Toast.LENGTH_LONG).show();  
    Toast.makeText(this, parser.getAttributeValue(null, "attr"), Toast.LENGTH_LONG)  
        .show();  
}
```

# Enregistrement dans un fichier plat



Il existe 2 façons d'écrire dans un fichier plat :

- Une sauvegarde interne dans la mémoire du téléphone.
- Une sauvegarde externe dans la carte SD de l'utilisateur.

Les informations enregistrées dans la **carte mémoire externe** du terminal mobile sont **accessibles** par l'ensemble des applications.

Les données sauvegardées en **mémoire interne** peuvent ou non être **partagé en lecture ou/et en écriture** avec les autres applications.

# Enregistrement dans un fichier plat



Voici la liste des différentes permissions pour la création d'un fichier dans la mémoire interne :

- **MODE\_PRIVATE** crée un fichier (ou remplace l'existant), le fichier ne sera disponible que pour notre application.
- **MODE\_APPEND** crée un fichier (concatène si le fichier existe).
- **MODE\_WORLD\_READABLE** accès en lecture par les autres applications.
- **MODE\_WORLD\_WRITEABLE** accès en écriture par les autres applications.
- **MODE\_WORLD\_READABLE/MODE\_WORLD\_WRITEABLE** accès en lecture et écriture par tous.

Pour l'écriture il faut utiliser la méthode **openfileoutput** qui va renvoyer un FileOutputStream. Pour la lecture il faut utiliser la méthode **openFileInput** qui renvoie un FileInputStream.

# DOM parser



le parser DOM permet de naviguer assez facilement dans la représentation arborescente du document XML. L'exemple suivant permet de chercher tous les tags "monitor" dans les sous-tags de la racine.

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
try {
    DocumentBuilder builder = factory.newDocumentBuilder();
    Document dom = builder.parse(source);
    Element root = dom.getDocumentElement();
    // Récupère tous les tags du descendant de la racine s'appelant monitor
    NodeList items = root.getElementsByTagName("monitor");
    for (int i=0;i<items.getLength();i++) {
        Node item = items.item(i);
        // Traitement:
        // item.getNodeName()
        // item.getTextContent() ...
    }
} catch (...) {}
```

Android dispose d'une base de données relationnelle basée sur SQLite.

Exemple de création de base de données, ce qui se fait en héritant de SQLiteOpenHelper:

```
public class DictionaryOpenHelper extends SQLiteOpenHelper {
    private static final int DATABASE_VERSION = 2;
    private static final String DICTIONARY_TABLE_NAME = "dictionary";
    private static final String DICTIONARY_TABLE_CREATE = "CREATE TABLE "
    + DICTIONARY_TABLE_NAME + " (" + KEY_WORD + " TEXT, " + KEY_DEFINITION + " TEXT);";

    DictionaryOpenHelper(Context context) {
        super(context, DATABASE_NAME, null, DATABASE_VERSION);
    }
    @Override
    public void onCreate(SQLiteDatabase db) {
        db.execSQL(DICTIONARY_TABLE_CREATE);
    }
}
```

# SQLite Lecture/Ecriture



Pour réaliser des écritures ou lectures, on utilise les méthodes **getWritableDatabase()** et **getReadableDatabase()** qui renvoient une instance de **SQLiteDatabase**. Sur cet objet, une requête peut être exécutée au travers de la méthode **query()** :

```
public Cursor query (boolean distinct, String table, String[] columns,
String selection, String[] selectionArgs, String groupBy,
String having, String orderBy, String limit)
```

# SQLite Lecture/Ecriture



L'objet de type **Cursor** permet de traiter la réponse (en lecture ou écriture).

- **getCount()**: nombre de lignes de la réponse
- **moveToFirst()**: déplace le curseur de réponse à la première ligne
- **getInt(int columnIndex)**: retourne la valeur (int) de la colonne passée en paramètre
- **getString(int columnIndex)**: retourne la valeur (String) de la colonne passée en paramètre
- **moveToNext()**: avance à la ligne suivante
- **getColumnName(int)**: donne le nom de la colonne désignée par l'index
- ...



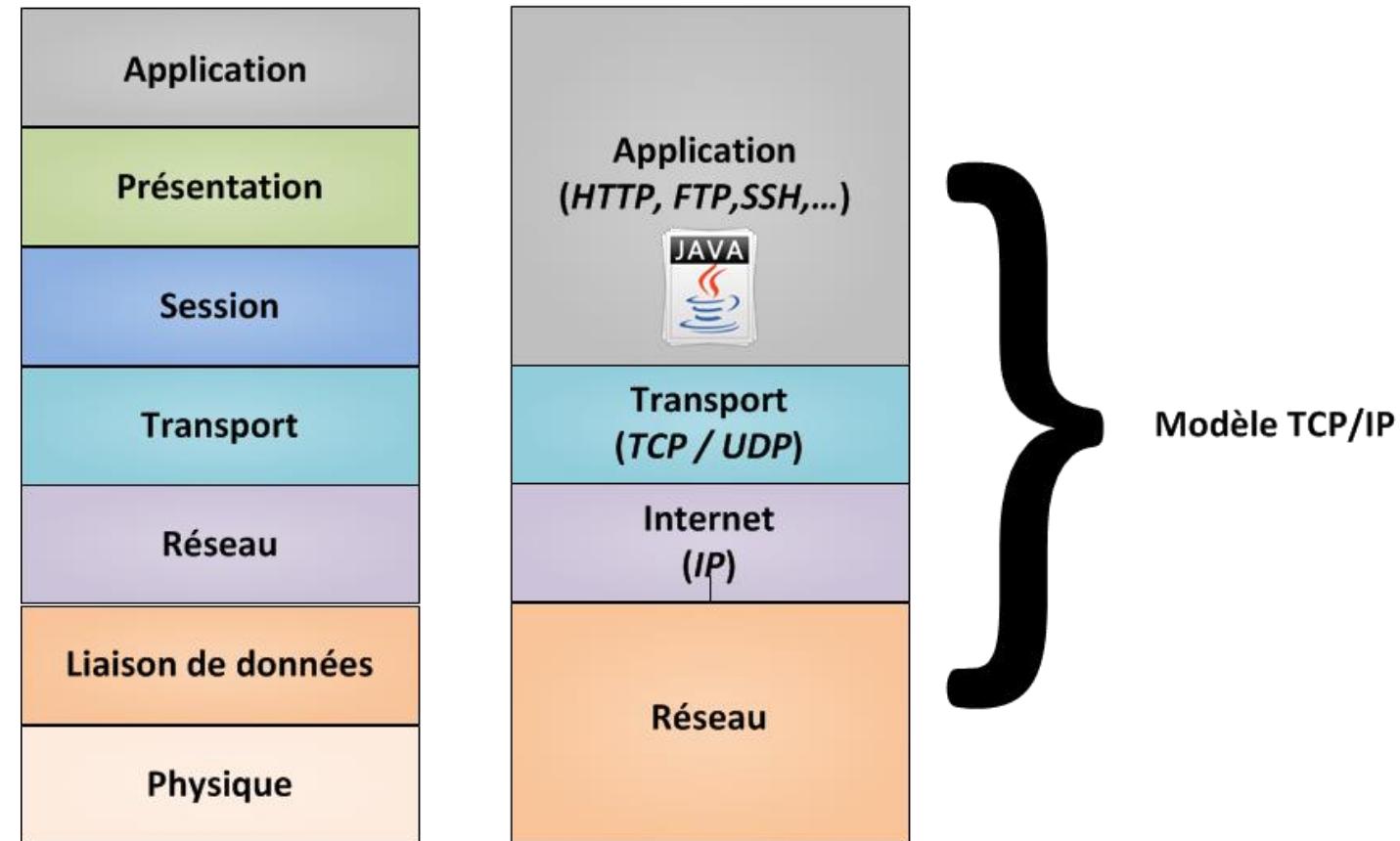
# La Gestion du Réseau

" Cycle Developpeur "

# La programmation réseau

Pour faire **communiquer deux machines sur le réseau** nous allons devoir utiliser des **adresses**, des **ports** et des **protocoles** mais pour que celles-ci puissent s'échanger des informations, les deux parties de la communication doivent utiliser des **sockets**.

Le modèle OSI et TCP:



# Les WebServices



Les webservices sont des services de traitement de la donnée exposés sur internet.

Ils fonctionnent sur un principe de contrat implémenté via un **fichier WSDL**.

Ce contrat expose des opérations, et décrit les formats de messages en attendues en entrée pour chaque opération.

Ces services peuvent être appelés à distance par des langages différents de celui avec lequel le web service a été codé.

Les web services ont pour intérêt majeur qu'ils **permettent de faire communiquer** appareils mobiles et serveurs d'application facilement.

Il existe 2 types de web services :

- **SOAP** : les web services classiques, en XML.
- **REST** : des web services XML, JSON, Text.

# Permissions



Pour pouvoir accéder à un web service, il faut donner le droit à l'application Android d'accéder à Internet.

On ajoutera dans le manifest la permission suivante :

```
<uses-permission android:name="android.permission.INTERNET" />
```

# WebServices REST



REST (**R**Epresentational **S**tate **T**ransfer) est un style d'architecture de services décrit pour la première fois par Roy Thomas Fielding.

REST définit donc un ensemble de contraintes de manière à optimiser certaines qualités telles que :

- La simplicité.
- Les performances réseaux.
- La généricité.
- La séparation des tâches.

JSON est souvent utilisé pour sérialiser et transmettre des données structurées au travers d'une connexion réseau.

Il est utilisé principalement pour transmettre des données entre un serveur et une application web, il sert d'alternative à XML.

## Schéma:

Les schémas Json sont destinés à fournir une validation, la documentation et le contrôle de l'interaction des données.

# JSON types de données



Les types basiques de Json sont :

- Les nombres.
- Les chaînes de caractère entre doubles quotes.
- Les booléens (true ou false),
- Les tableaux : une série ordonnée de valeurs séparées,
- Les objets : une collection non ordonnée de clés.

Il existe des types de données natifs non supportés par Json tels que Date, Error, Function ou Regular Expression.

Ces types de données JavaScript doivent être représentés à l'aide d'autres formats.

# Comparaison avec XML/JSON



	Qualités	Point faible
JSON	Très répandu Bien outillé Peux verbeux	Pas de format de définition existant
XML	Très répandu Lisible Très outillé Possède des langages de description : DTD ou XML Schéma	Parfois complexe à mettre en œuvre Très verbeux

# JSON types de données



Le SDK d'Android fournit déjà deux APIs qui permettent d'envoyer des requêtes à des serveurs webs :

- HttpURLConnection
- Apache HTTP Client -> qui est déprécié depuis l'API niveau 22.

Nous utiliserons **HttpURLConnection** ainsi la librairie Gson. Il faut donc ajouter le jar Gson\*.jar au projet au travers du **build path**.

Gson (aussi connu sous le nom de **Google Gson**) est une librairie **open-source Java** qui permet de **sérialiser et déserialiser des objets Java** en Json.

# Appel aux WebServices



Pour effectuer les appels il faut avoir les éléments suivants dans la classe d'appel :

- La base de l'url pour le webservice
- L'objet Gson
- Un constructeur
- Une méthode privée pour se connecter en Http au WebService et récupérer un inputStream
- Des méthodes pour chaque API disponible

Exemple de web service qui récupère une liste de catégorie d'articles (**HttpURLConnection**) :

```
lUrl = new URL(ConfigServeur.ADRESSE_IP_CONNEXION +"/project-exemple/webservice/listCategorie");
lURLConnection = (HttpURLConnection) lUrl.openConnection();
InputStream in = new BufferedInputStream(lURLConnection.getInputStream());
```

# Appel aux WebServices



La librairie GSon permet de transformer un inputStream en Objet et ce, en une seule ligne de code:

```
Gson gson = new Gson();
NdfCategorie[] liste = gson.fromJson(new InputStreamReader(in), NdfCategorie[].class);
```

# AsyncTask



Les AsyncTask permettent une utilisation correcte et facile du ThreadUI. Cette classe permet **d'effectuer des tâches de fond et de publier des résultats sans manipuler des threads et/ou des handlers.**

Une tâche asynchrone est définie par 3 types génériques, appelés :

- le paramètre
- la progression
- le résultat

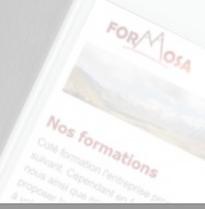
Ainsi que 4 étapes :

- OnPreExecute
- doInBackground
- onProgressUpdate
- onPostExecute.

Idéalement, les AsyncTasks sont utilisées pour des **utilisations courtes**.

AsyncTask est une classe abstraite. Pour l'utiliser, il faut donc créer une classe qui en hérite.

# Utilisation



```
class DownloadFilesTask extends AsyncTask<URL, Integer, Long> {
    protected Long doInBackground(URL... urls) {
        int count = urls.length;
        long totalSize = 0;
        for (int i = 0; i < count; i++) {
            totalSize += Downloader.downloadFile(urls[i]);
            publishProgress((int) ((i / (float) count) * 100));
            // Escape early if cancel() is called
            if (isCancelled()) break;
        }
        return totalSize;
    }

    protected void onProgressUpdate(Integer... progress) {
        setProgressPercent(progress[0]);
    }
    protected void onPostExecute(Long result) {
        showDialog("Downloaded " + result + " bytes");
    }
}
```

# AsyncTask



Une fois créée, une tâche est exécutée très simplement:

```
new DownloadFilesTask().execute(url1, url2, url3);
```

De cette manière, les tâches sont mises dans une file d'attente et s'exécuteront les unes après les autres.

Pour quelles s'exécutent parallèlement :

```
new DownloadFilesTask().execute(AsyncTask.THREAD_POOL_EXECUTOR, url1,url2,url3);
```

# Les types génériques



Les trois types utilisés par une tâche asynchrone sont les suivants:

- **Params**, le type des paramètres envoyés à la tâche lors de l'exécution.
- **Progrès**, le type des unités d'étape publiées lors du calcul de fond.
- Par conséquent, le **type de résultat** du calcul en arrière-plan.

Tous les types ne sont pas toujours utilisés par une tâche asynchrone. Pour marquer un type non utilisé, il suffit d'utiliser le type Void:

```
class MaTache extends AsyncTask<Void, Void, Void> { ... }
```

# Cycle de vie



Lorsqu'une tâche asynchrone est exécutée, la tâche passe par 4 étapes:

- **OnPreExecute ()**, Cette étape est normalement utilisée pour configurer la tâche, par exemple, en montrant une barre de progression dans l'interface utilisateur.
- **doInBackground (Param ...)**, Cette étape est utilisée pour effectuer le calcul de fond qui peut prendre un certain temps.
- **onProgressUpdate (Progress ...)**, Cette méthode est utilisée pour afficher toute forme de progression dans l'interface utilisateur tandis que le calcul de fond est toujours en cours d'exécution.
- **onPostExecute (Résultat)**, appelé sur le thread UI après la fin du calcul. Le résultat est transmis à cette étape en tant que paramètre.

# Règles



Pour que cette classe fonctionne correctement, il y a quelques règles à respecter :

- La classe AsyncTask doit être chargée sur le thread UI. Ceci est fait automatiquement dès JELLY\_BEAN (API 16, 17, 18).
- L'instance de la tâche doit être créée sur le thread UI.
- execute(Param ...) doit être invoqué sur le thread UI.
- Ne pas appeler OnPreExecute (), onPostExecute (Résultat), doInBackground(Param ...), onProgressUpdate (Progress ...) manuellement.
- La tâche ne peut être exécutée qu'une seule fois (une exception sera levée si une deuxième exécution est tentée).



# Les Compléments

" Cycle Developpeur "



Quand on crée un objet, on réserve dans la mémoire allouée par le processus un emplacement qui aura la place nécessaire pour mettre l'objet. Pour accéder à l'objet, on utilise une référence sous forme d'un identifiant déclaré dans le code :

```
String chaine = new String();
```

Ici, *chaine* est l'identifiant, autrement dit une référence qui pointe vers l'emplacement mémoire réservé pour cette chaîne de caractères.

Bien sûr, au fur et à mesure que le programme s'exécute, on va allouer de la place pour d'autres objets et, si on ne fait rien, la mémoire va être saturée.

Afin de faire en sorte de libérer de la mémoire, un processus qui s'appelle le GC (GC, « ramasse-miettes » en français) va détruire les objets qui ne sont plus susceptibles d'être utilisés :

```
String chaine = new String("Rien du tout");

if(chaine.equals("Quelque chose")) {
    int dix = 10;
}
```

# Référence forte et faible



La variable chaine sera disponible avant, pendant et après le if puisqu'elle a été déclarée avant (donc de 1 à 5, voire plus loin encore), en revanche dix a été déclaré dans le if, il ne sera donc disponible que dedans (donc de 4 à 5). Dès qu'on sort du if, le GC passe et dés alloue la place réservée dix de manière à pouvoir l'allouer à un autre objet.

Quand on crée un objet en Java, il s'agit toujours d'une référence forte, c'est-à-dire que l'objet est protégé contre le GC tant qu'on est certain que vous l'utilisez encore.

A l'inverse, les références faibles ne protège pas une référence du GC. Si vous avez une référence forte vers un objet, le GC ne passera pas dessus.

Si vous en avez deux, idem.

Si le GC réalise que l'une des deux références fortes n'est plus valide, l'objet est toujours conservé en mémoire puisqu'il reste une référence forte.

En revanche, dès que la seconde référence forte est invalidée, alors l'espace mémoire est libéré puisqu'il ne reste plus aucune référence forte.

# Référence forte et faible



Il suffit d'inclure une référence faible vers notre activité dans l'AsyncTask pour pouvoir garder une référence vers l'activité sans pour autant la protéger contre le GC.

Pour créer une référence faible d'un objet T, on utilise WeakReference de cette manière :

```
T strongReference = new T();
WeakReference<T> weakReference = new
WeakReference<T>(strongReference);
```

Il n'est pas possible d'utiliser directement un *WeakReference*, comme il ne s'agit que d'une référence faible, il vous faut donc récupérer une référence forte de cet objet.

Pour ce faire, il suffit d'utiliser T *get()*. Cependant, cette méthode renverra **null** si l'objet a été nettoyé par le GC.

# Gestion de la téléphonie - Appels



Sur Android, pour pouvoir passer un appel depuis une application, il suffit de lancer une intention vers l'application de gestion d'appel d'Android.

Il faut cependant envoyer à cette intention les données d'appel (Le numéro de téléphone).

## Exemple:

```
Intent phoneCallIntent = new Intent(Intent.ACTION_CALL);
phoneCallIntent.setData(Uri.parse("numéro_de_téléphone"));
pContext.startActivity(phoneCallIntent);
```

# Gestion de la téléphonie - Sms



Pour un sms depuis une application, on procède de la même manière, on lance une intention vers l'application de gestion de messages.

On va définir :

- Le **type** d'action : **vnd.android-dir/mms-sms**
- Le **address** : le numéro de téléphone
- Le **sms\_body**: le corps du message.

**Exemple:**

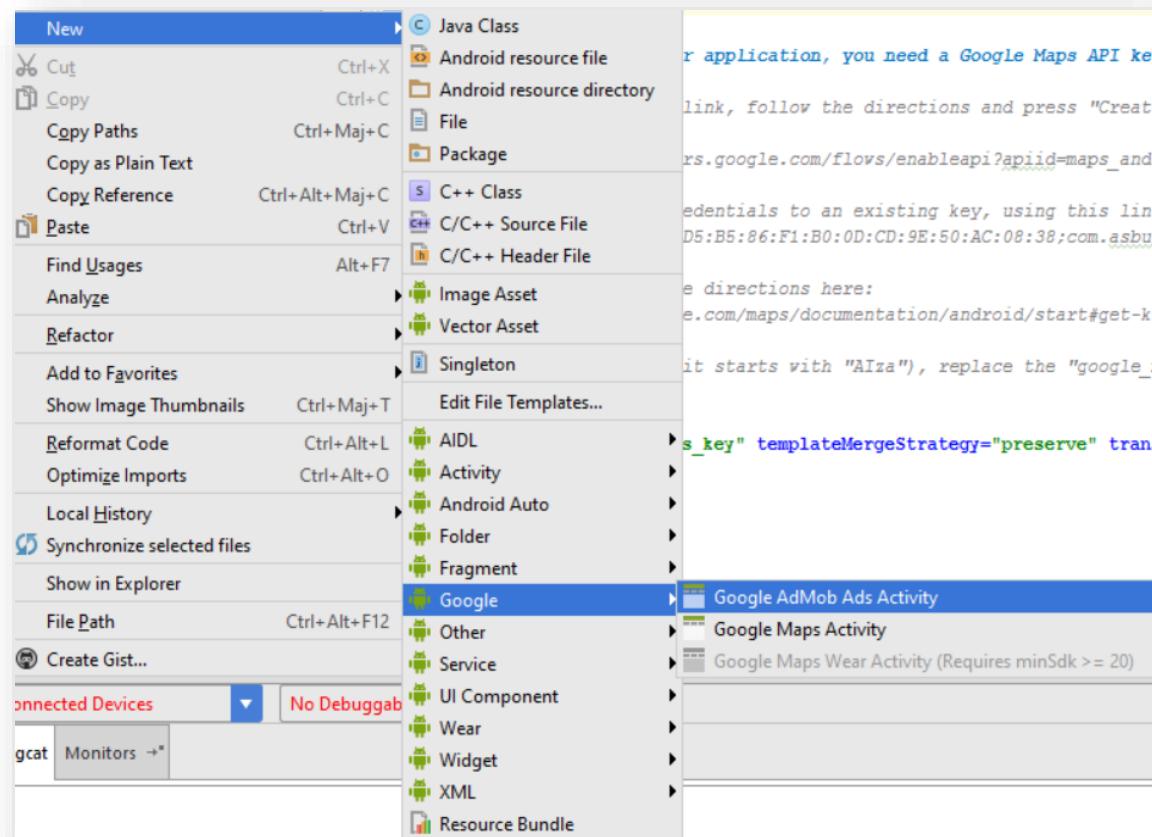
```
Intent smsIntent = new Intent(android.content.Intent.ACTION_VIEW);
smsIntent.setType("vnd.android-dir/mms-sms");
smsIntent.putExtra("address", "your desired phoneNumber");
smsIntent.putExtra("sms_body", "your desired message");
startActivity(smsIntent);
```

# GoogleMap



Pour utiliser l'API google map, il faut d'abord créer une nouvelle activité Google Maps Activity.

Android Studio ouvre les fichiers **google\_maps\_api.xml** et **MapsActivity.java** dans l'éditeur.



# GoogleMap clé API

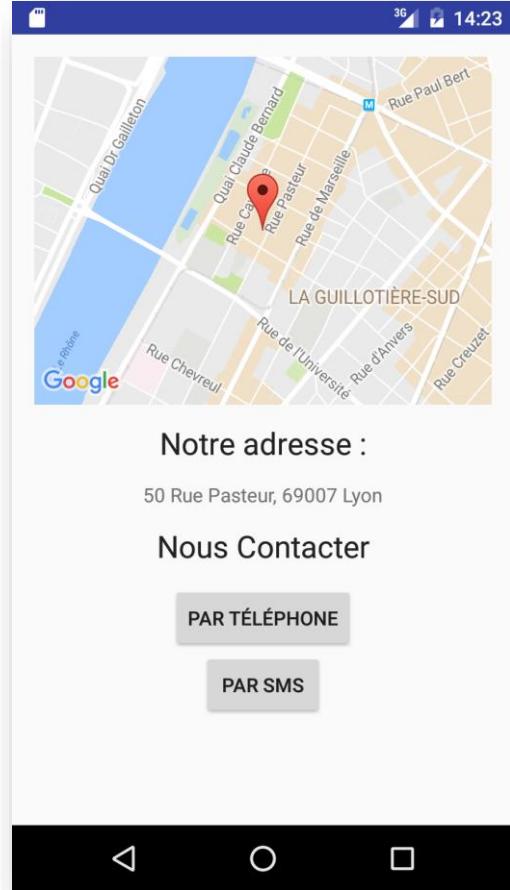


Pour obtenir un clé API le plus simple et le plus rapide :

Utilisez le lien fourni dans le fichier `google_maps_api.xml` qu'Android Studio a créé pour vous :

1. Copiez le lien fourni dans le fichier **`google_maps_api.xml`** et collez-le dans votre navigateur.  
Suivez les instructions pour créer un nouveau projet sur la console ou sélectionnez un projet existant.
2. Créez **une clé d'API Android** pour votre projet de console.
3. Copiez **la clé d'API obtenue**, retournez dans Android Studio et collez la clé d'API dans l'élément `<string>` du fichier **`google_maps_api.xml`**.

# TP 20 – Google map



Réalisation d'une application géolocalisée présentée à l'aide de Google Maps.

# Personnalisation de View

Il est possible de se personnaliser un composant graphique en spécifiant dans le XML le lien vers la classe.

Prenons par exemple le composant Surface, nous pouvons créer une Classe MySurface et changer l'élément Surface par la package et la classe.

```
public class MySurface extends SurfaceView{  
  
    public MySurface(Context context, AttributeSet attrs) {  
        super(context, attrs);  
    }  
}
```

```
<com.example.axopen.exempleapplication.MySurface  
    android:layout_width="match_parent"  
    android:layout_height="match_parent"  
    android:id="@+id/surfaceView"  
    android:layout_alignParentTop="true"  
    android:layout_centerHorizontal="true"  
    android:background="@color/background_floating_material_light" />
```

# Dessin avec un canvas

Il est possible de dessiner la vue avec un Canvas.

La méthode invalidate va servir à redessiner la vue.

Ici nous changeons la couleur du carré à chaque Click. Attention il ne faut pas oublier d'ajouter le listener dans le constructeur de la vue

```
public class MySurface extends SurfaceView
    implements View.OnClickListener {

    Paint myPaint = new Paint();

    ...

    @Override
    public void onDraw(Canvas c) {
        super.onDraw(c);
        myPaint.setStrokeWidth(1);
        c.drawRect(20, 20, 40, 40, myPaint);
    }

    @Override
    public void onClick(View v) {
        myPaint.setColor(Color.rgb(
            (int)(Math.random() * 255),
            (int)(Math.random() * 255),
            (int)(Math.random() * 255)));
        this.invalidate();
    }
}
```

# Les animations

Il est possible d'animer les composants (View) par la fonction **animate()**

Une animation peut être vue comme une trajectoire dans l'espace d'une variable visuelle.

Si nous prenons la position x et y, une trajectoire dans l'espace X, Y provoquera un déplacement

Si nous prenons la taille, une trajectoire dans cet espace provoquera un agrandissement ou une diminution.

Etc.

```
Button but2 = (Button) findViewById(R.id.button6);
but2.animate()
    .translationX(30)
    .alpha(0f)
    .setDuration(3000);
```