

# Historique des versions

Nom	Version	Date
<b>Android</b>	<b>1.0</b>	<b>09/2008</b>
<b>Petit Four</b>	<b>1.1</b>	<b>02/2009</b>
<b>Cupcake</b>	<b>1.5</b>	<b>04/2009</b>
<b>Donut</b>	<b>1.6</b>	<b>09/2009</b>
<b>Gingerbread</b>	<b>2.3</b>	<b>12/2010</b>
<b>Honeycomb</b>	<b>3.0</b>	<b>02/2011</b>
<b>Ice Cream Sandwich</b>	<b>4.0.1</b>	<b>10/2011</b>
<b>Jelly Bean</b>	<b>4.1</b>	<b>07/2012</b>
<b>KitKat</b>	<b>4.4</b>	<b>10/2013</b>
<b>Lollipop</b>	<b>5.0</b>	<b>10/2014</b>
<b>Marshmallow</b>	<b>6.0</b>	<b>05/2015</b>
<b>Nougat</b>	<b>7.0</b>	<b>09/2016</b>
<b>Oreo</b>	<b>8.0</b>	<b>08/2017</b>

# Android

- L'écosystème d'Android s'appuie sur deux piliers:
  - le langage Java ou Kotlin
  - le SDK qui permet d'avoir un environnement de développement facilitant la tâche du développeur
- Le kit de développement donne accès à des exemples, de la documentation mais surtout à l'API de programmation du système et à un émulateur pour tester ses applications.
- Stratégiquement, Google utilise la licence Apache pour Android ce qui permet la redistribution du code sous forme libre ou non et d'en faire un usage commercial.
- Le SDK était:
  - **anciennement** manipulé par un plugin d'Eclipse (obsolète)
  - **maintenant** intégré à Android Studio (IntelliJ)

# ***L'environnement Android Studio***

Depuis mi-2015, il faut utiliser Android Studio (IDE IntelliJ ayant subi l'intégration de fonctionnalités de développement Android).

## **Avantages**

- meilleur intégration du SDK dans Android Studio
- puissance de l'IDE IntelliJ

## **Desavantages**

- lourdeur de l'IDE IntelliJ
- moins d'outils standalone (gestion des émulateurs, du SDK)
- obligation de s'habituer à un autre IDE
- nouvelle architecture des répertoires
- gestion des dépendances avec gradle

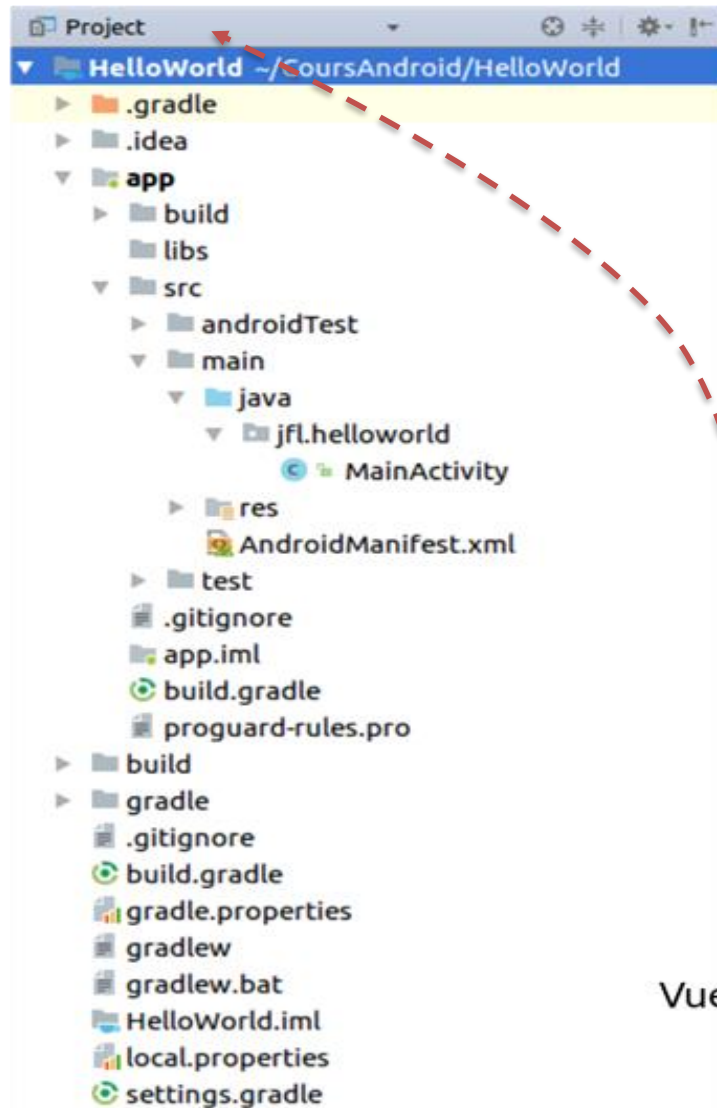
# Développement Android

Le but de ce cours est de découvrir la programmation sous Android, sa plate-forme de développement et les spécificités du développement embarqué sur *Smartphone*.

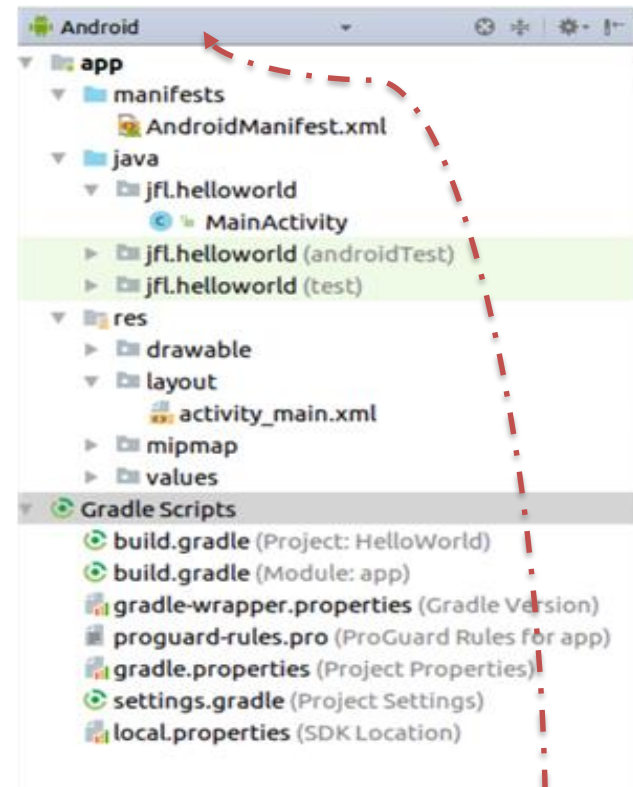
# ***L'architecture d'un projet Android Studio***

- app: le code de votre application
- build: le code compilé
- lib: les librairies natives
- src: les sources de votre applications
- main/java: vos classes
- main/res: vos ressources (XML, images, ...)
- test: les tests unitaires
- Des fichiers de configuration:
  - build.gradle (2 instances): règles de dépendance et de compilation
  - settings.gradle: liste de toutes les applications à compiler (si plusieurs)

# Attention aux vues dans Android Studio



Vue Project



Vue Android

# ***Le Manifest de l'application***

Le fichier **AndroidManifest.xml** déclare l'ensemble des éléments de l'application

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="andro.jf"
    android:versionCode="1"
    android:versionName="1.0">
    <application android:icon="@drawable/icon"
        android:label="@string/app_name">

        <activity android:name=".Main"
            android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>

        <service>...</service>
        <receiver>...</receiver>
        <provider>...</provider>

    </application>
</manifest>
```

# Les ressources

Les ressources de l'application sont utilisées dans le code au travers de la classe statique **R**.

ADT re-génère automatiquement la classe statique **R** à chaque changement dans le projet. Toutes les ressources sont accessibles au travers de **R**, dès qu'elles sont déclarées dans le fichier XML ou que le fichier associé est déposé dans le répertoire adéquat. Les ressources sont utilisées de la manière suivante:

```
android.R.type_ressource.nom_ressource
```

qui est de type int. Il s'agit en fait de l'identifiant de la ressource. On peut alors utiliser cet identifiant ou récupérer l'instance de la ressource en utilisant la classe Resources:

```
Resources res = getResources();  
String hw = res.getString(R.string.hello);  
XXX o = res.getXXX(id);
```

Une méthode spécifique pour les objets graphiques permet de les récupérer à partir de leur id, ce qui permet d'agir sur ces instances même si elles ont été créées via leur définition XML:

```
TextView texte = (TextView)findViewById(R.id.le_texte);  
texte.setText("Here we go !");
```



## Les chaines

Les chaines constantes de l'application sont situées dans **res/values/strings.xml**. L'externalisation des chaines permettra de réaliser l'internationalisation de l'application. Voici un exemple:

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <string name="hello">Hello Hello JFL !</string>
    <string name="app_name">AndroJF</string>
</resources>
```


La récupération de la chaine se fait via le code:

```
Resources res = getResources();
String hw = res.getString(R.string.hello);
```

### Android Resources (default)

#### Resources Elements

        Az

 hello (String)

 app\_name (String)


Add...

Remove...

Up

Down

#### Attributes for hello (String)

 Strings, with optional simple formatting, can be stored and retrieved as resources. You can add formatting to your string by using three standard HTML tags: b, i, and u. If you use an apostrophe or a quote in your string, you must either escape it or enclose the whole string in the other kind of enclosing quotes.

Name\* hello

Value\* Hello Hello JFL !

# Internationalisation du projet

Voir tutoriel

# Les activités

Une application Android étant hébergée sur un système embarqué, le cycle de vie d'une application ressemble à celle d'une application Java ME. L'activité peut passer des états:

- démarrage -> actif: détient le focus et est démarré
- actif -> suspendue: ne détient plus le focus
- suspendue -> actif:
- suspendue -> détruit:

Le nombre de méthodes à surcharger et même plus important que ces états:

```
public class Main extends Activity {  
    public void onCreate(Bundle  
        savedInstanceState) {  
        super.onCreate(savedInstanceState);  
        setContentView(R.layout.acceuil); }  
    protected void onDestroy() {  
        super.onDestroy(); }  
    protected void onPause() {  
        super.onPause(); }  
    protected void onResume() {  
        super.onResume(); }  
    protected void onStart() {  
        super.onStart(); }  
        protected void onStop() {  
            super.onStop(); } }  
}
```

# Les activités

## ***CYCLE DE VIE D'UNE ACTIVITÉ***

- **onCreate()** / **onDestroy()**: permet de gérer les opérations à faire avant l'affichage de l'activité, et lorsqu'on détruit complètement l'activité de la mémoire. On met en général peu de code dans **onCreate()** afin d'afficher l'activité le plus rapidement possible.
- **onStart()** / **onStop()**: ces méthodes sont appelées quand l'activité devient visible/invisible pour l'utilisateur.
- **onPause()** / **onResume()**: une activité peut rester visible mais être mise en pause par le fait qu'une autre activité est en train de démarrer, par exemple B. **onPause()** ne doit pas être trop long, car B ne sera pas créé tant que **onPause()** n'a pas fini son exécution.
- **onRestart()**: cette méthode supplémentaire est appelée quand on relance une activité qui est passée par **onStop()**. Puis **onStart()**
- est aussi appelée. Cela permet de différencier le premier lancement d'un relancement.

# Les activités

## ***Sauvegarde des interfaces d'activité***

L'objet **Bundle** passé en paramètre de la méthode **onCreate** permet de restaurer les valeurs des interfaces d'une activité qui a été déchargée de la mémoire. En effet, lorsque l'on appuie par exemple sur la touche *Home*, en revenant sur le bureau, Android peut-être amené à déchargé les éléments graphiques de la mémoire pour gagner des ressources. Si l'on rebascule sur l'application (appui long sur *Home*), l'application peut avoir perdu les valeurs saisies dans les zones de texte.

Pour forcer Android à décharger les valeurs, il est possible d'aller dans "Development tools > Development Settings" et de cocher "Immediately destroy activities".

**Si une zone de texte n'a pas d'identifiant, Android ne pourra pas la sauver et elle ne pourra pas être restaurée à partir de l'objet Bundle.**

Si l'application est complètement détruite (tuée), rien n'est restauré.

Le code suivant permet de visualiser le déclenchement des sauvegardes:

```
protected void onSaveInstanceState(Bundle outState) {  
    super.onSaveInstanceState(outState);  
    Toast.makeText(this, "Sauvegarde !", Toast.LENGTH_LONG).show();  
}
```

# **Interfaces graphiques**

# Vues et gabarits

- Les éléments graphiques héritent de la classe View. On peut regrouper des éléments graphiques dans une ViewGroup. Des
- ViewGroup particuliers sont prédéfinis: ce sont des gabarits (*layout*) qui proposent une prédispositions des objets graphiques:
- LinearLayout: dispose les éléments de gauche à droite ou du haut vers le bas
- RelativeLayout: les éléments enfants sont placés les uns par rapport aux autres
- TableLayout: disposition matricielle
- FrameLayout: disposition en haut à gauche en empilant les éléments
- GridLayout: disposition matricielle avec N colonnes et un nombre infini de lignes

Les déclarations se font principalement en XML, ce qui évite de passer par les instanciations Java.

- ***Attributs des gabarits***
- Les attributs des gabarits permettent de spécifier des attributs supplémentaires. Les plus importants sont:
- android:layout\_width et android:layout\_height:
  - ="match\_parent": l'élément remplit tout l'élément parent
  - ="wrap\_content": prend la place minimum nécessaire à l'affichage
  - ="fill\_parent": comme match\_parent (deprecated, API<8)
- android:orientation: définit l'orientation d'empilement
- android:gravity: définit l'alignement des éléments

**Voir Exemple**

# Inclusions de gabarits

Les interfaces peuvent aussi inclure d'autres interfaces, permettant de factoriser des morceaux d'interface. On utilise dans ce cas le mot clef **include**

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    >
    <include android:id="@+id/include01"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        layout="@layout/acceuil"
        ></include>
</LinearLayout>
```

**Suite interface graphique : positionnement avancé**



# Les Intents

# Principe des Intents

- Les *Intents* permettent de gérer l'envoi et la réception de messages afin de faire coopérer les applications. Le but des *Intents* est de déléguer une action à un autre composant, une autre application ou une autre activité de l'application courante.
- Un objet **Intent** contient les informations suivantes:
  - le nom du composant ciblé (facultatif)
  - l'action à réaliser, sous forme de chaîne de caractères
  - les données: contenu MIME et URI
  - des données supplémentaires sous forme de paires clef/valeur
  - une catégorie pour cibler un type d'application
  - des drapeaux (informations supplémentaires)
- On peut envoyer des *Intents* informatifs pour faire passer des messages. Mais on peut aussi envoyer des *Intents* servant à lancer une nouvelle activité.

# Intents pour une nouvelle activité

Il y a plusieurs façons de créer l'objet de type *Intent* qui permettra de lancer une nouvelle activité. Si l'on passe la main à une activité interne à l'application, on peut créer l'Intent et passer la classe de l'activité ciblée par l'Intent:

```
Intent new = new Intent(getApplicationContext(), ClassCible.class);
startActivity(new);
```

Le premier paramètre de construction de l'Intent est en fait le contexte de l'application. Dans certain cas, il ne faut pas mettre **this** mais faire appel à **getApplicationContext()** si l'objet manipulant l'*Intent* n'hérite pas de **Context**.

S'il s'agit de passer la main à une autre application, on donne au constructeur de l'Intent les données et l'URI cible: l'OS est chargé de trouver une application pouvant répondre à l'Intent.

**Sans oublier de déclarer la nouvelle activité dans le Manifest.**

## ***Retour d'une activité***

- Lorsque le bouton *retour* est pressé, l'activité courante prend fin et revient à l'activité précédente. Cela permet par exemple de terminer son appel téléphonique et de revenir à l'activité ayant initié l'appel.
- Au sein d'une application, une activité peut vouloir récupérer un code de retour de l'activité "enfant". On utilise pour cela la méthode **startActivityResult** qui envoie un code de retour à l'activité enfant. Lorsque l'activité parent reprend la main, il devient possible de filtrer le code de retour dans la méthode **onActivityResult** pour savoir si l'on revient ou pas de l'activité enfant.
- L'appel d'un *Intent* devient donc:

```
public void onCreate(Bundle savedInstanceState) {
    Intent login = new Intent(getApplicationContext(), GivePhoneNumber.class);
    startActivityForResult(login, 48);
    ...
}
```

Le filtrage dans la classe parente permet de savoir qui avait appelé cette activité enfant:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (requestCode == 48)
        Toast.makeText(this, "Code de requête récupéré (je sais d'ou je viens)",
            Toast.LENGTH_LONG).show();
}
```

# Résultat d'une activité

Il est aussi possible de définir un résultat d'activité, avant d'appeler explicitement la fin d'une activité avec la méthode **finish()**. Dans ce cas, la méthode **setResult** permet d'enregistrer un code de retour qu'il sera aussi possible de filtrer dans l'activité parente.

Dans l'activité enfant, on met donc:

```
Button finish = (Button) findViewById(R.id.finish);
finish.setOnClickListener(new OnClickListener() {

    @Override
    public void onClick(View v) {
        setResult(50);
        finish();
    }
});
```

Et la classe parente peut filtrer ainsi:

```
protected void onActivityResult(int requestCode, int resultCode, Intent data)
{
    if (requestCode == 48)
        Toast.makeText(this, "Code de requête récupéré (je sais d'ou je viens)",
            Toast.LENGTH_LONG).show();
    if (resultCode == 50)
        Toast.makeText(this, "Code de retour ok (on m'a renvoyé le bon code)",
            Toast.LENGTH_LONG).show();
}
```

# Ajouter des informations

- Les *Intent* permettent de transporter des informations à destination de l'activité cible. On appelle ces informations des *Extra*: les méthodes permettant de les manipuler sont **getExtra** et **putExtra**. Lorsqu'on prépare un *Intent* et que l'on souhaite ajouter une information de type "clef -> valeur" on procède ainsi:

```
Intent callactivity2 = new Intent(getApplicationContext(),  
Activity2.class); callactivity2.putExtra("login", "jfl");  
startActivity(callactivity2);
```

Du côté de l'activité recevant l'Intent, on récupère l'information de la manière suivante:

```
Bundle extras = getIntent().getExtras();  
String s = new String(extras.getString("login"));
```

## Voir exemple

- D'autres actions permettent de lancer des applications tierces pour déléguer un traitement:

- **ACTION\_CALL (ANSWER, DIAL):**  
passer/réceptionner/afficher un appel

- **ACTION\_SEND:** envoyer des données par SMS ou E-mail

- **ACTION\_WEB\_SEARCH:** rechercher sur internet

- Pour plus de détails voir site:

*<https://developer.android.com/reference/android/content/Intent>*