جامعة محمد الأول بوجدة
UNIVERSITÉ MOHAMMED PREMIER OUJDA

المدرسة الوطنية للذكاء الاصطناعي والرقمنة - بركان
ÉCOLE NATIONALE DE L'INTELLIGENCE ARTIFICIELLE ET DU DIGITAL - BERKANE

DEVELOPPEMENT D'APPLICATION MOBILE

2ÈME ANNÉE DU CYCLE INGÉNIEUR – ENIADB
Filière : GI – ROC - IA

Prof. Mohamed BOUDCHICHE
Email : m.boudchiche0@gmail.com

Année universitaire 2024-2025

OBJECTIFS DU COURS

- Comprendre les caractéristiques et spécificités des différents systèmes d'exploitation mobiles.
- Identifier les contraintes techniques liées au développement mobile.
- Développer des applications sur la plateforme Android.
- Concevoir des interfaces utilisateur intuitives pour les applications mobiles.
- Intégrer et utiliser des API REST dans les applications mobiles.
- Maîtriser le développement multiplateforme avec Flutter.

PLAN DU COURS

- Introduction aux Systèmes d'Exploitation Mobiles.
- Développement d'Applications Android.
- Conception d'Interfaces Graphiques Mobiles.
- Utilisation d'API REST dans les Applications Mobiles.
- Développement Multiplateforme avec Flutter.

Contexte et Évolution des Appareils Mobiles

- Les appareils mobiles ont évolué pour devenir des outils essentiels de communication, de divertissement et de travail.
  - juillet 2011: 550 000 utilisateurs actifs
  - avril 2013: 1.5 millions utilisateurs actifs
  - Août 2017: 2 billions d'utilisateurs actifs
  - Avril 2024: 5.65 billions d'utilisateurs actifs.
- De simples téléphones à des smartphones puissants avec des millions d'applications disponibles.

Popularité des Applications Mobiles

- Les applications mobiles sont devenues indispensables dans la vie quotidienne.

Les Objectifs des Applications Mobiles pour les Entreprises

- Élargir la portée des services ou produits.
- Améliorer l'engagement client avec des services instantanés.
- Optimiser l’expérience utilisateur.
- Générer des revenus via des applications freemium ou payantes.

Exemples d'Applications Mobiles Réussies

- **Uber :** Application de transport multiplateforme.
- **Instagram :** Application native sur iOS et Android, forte performance et UX.
- **Spotify :** Application hybride avec un accès natif aux fonctionnalités audio.

01

Introduction aux Systèmes d'Exploitation Mobiles.

Introduction aux Systèmes d'Exploitation Mobiles

- Vue d'ensemble des systèmes d'exploitation (Android, iOS, autres)
- Comparaison des architectures des systèmes d'exploitation mobiles
- Écosystème mobile et tendances actuelles
- Environnements de développement (Android Studio, Xcode, etc.)

Vue d'ensemble des Systèmes d'Exploitation Mobiles

- Android (Google)
- iOS (Apple)
- HarmonyOS (Huawei)
- KaiOS (feature phones)

Android (Google)

- Open-source, basé sur Linux
- Grande diversité des appareils (Samsung, Xiaomi, Google Pixel, etc.)
- Personnalisation par les fabricants (interfaces customisées)
- Fragmentation du système d'exploitation
- Versions : Android 15 (dernier : 3 septembre 2024)

iOS (Apple)

- Système fermé et optimisé pour l'écosystème Apple
- App Store uniquement, environnement sécurisé
- Mises à jour simultanées sur tous les appareils compatibles
- Versions : iOS 18.0.1 (dernier : 3 octobre 2024)

HarmonyOS (Huawei)

- Développé par Huawei suite aux restrictions américaines
- Système distribué pour smartphones, tablettes, objets connectés
- Pas de services Google, utilisation de l'AppGallery
- Principalement adopté en Chine
- Versions : HarmonyOS 4 (dernier)

KaiOS (feature phones)

- Système léger pour les téléphones basiques (feature phones)
- Accès à des applications comme WhatsApp, YouTube, Facebook
- Particulièrement populaire dans les marchés émergents
- Conçu pour des téléphones à faible coût et à longue autonomie
- Versions : KaiOS 3 (dernier)

Comparaison des systèmes d'exploitation mobiles

Système d'exploitation	Part de marché mondiale	Modèle économique	Appareils cibles	Caractéristiques principales
Android	~70%	Open-source	Smartphones, tablettes	Personnalisable, fragmentation
iOS	~27%	Fermé	iPhone, iPad	Optimisation, sécurité
HarmonyOS	Limité	Fermé (Huawei)	Smartphones Huawei	Multi-appareils, sans Google
KaiOS	Limité	Fermé	Feature phones	Simple, longévité de la batterie

Comparaison des Architectures des Systèmes d'Exploitation

- Architecture basée sur le noyau Linux (Android)
- Architecture propriétaire et optimisée (iOS)
- Modèle de développement : ouvert (Android) vs fermé (iOS)

Tendances Actuelles dans le Développement Mobile

- Montée en puissance des Progressive Web Apps (PWA)
- Développement multiplateforme (Flutter, React Native)
- Importance croissante de la sécurité mobile

Introduction aux Environnements de Développement Mobile

- Android Studio (Google)
- Xcode (Apple)
- Outils multiplateformes (Flutter, React Native)

02

Développement d'Applications Android.

19

L'écosystème Android

- L'écosystème d'Android s'appuie sur deux piliers principaux :
  - Java : Le langage de programmation traditionnel pour Android.
  - SDK Android : Fournit un environnement de développement complet et des outils pour faciliter le travail des développeurs.

Le Kit de Développement Android (SDK)

- Le kit de développement inclut :
  - Exemples de code et documentation détaillée.
  - API de programmation du système Android.
  - Émulateur Android pour tester les applications sur des dispositifs virtuels.

Stratégie de Google avec la Licence Apache

- Google utilise la licence Apache pour Android :
  - Redistribution du code sous forme libre ou non.
  - Utilisation commerciale autorisée.

Le Plugin *Android Development Tool* (ADT)

- ADT permet d'intégrer les fonctionnalités du SDK à Eclipse.
  - Installation sous forme de plugin.
  - Configuration du chemin du SDK.
  - Utilisable également dans Android Studio.

Architecture de l'OS Android

- Android est un système d'exploitation basé sur Linux, mais sans les outils GNU.

- Il repose sur plusieurs couches, dont :
  - Noyau Linux : Gère les opérations de bas niveau et les pilotes matériels.
  - Couche d'abstraction matérielle (HAL) : Interface entre le matériel et le logiciel.
  - Machine virtuelle Dalvik (avant Android 5.0 Lollipop) ou ART (Android Runtime) après Lollipop.

Les Bibliothèques et API dans Android

- **Android inclut plusieurs bibliothèques essentielles, telles que :**
  - SSL : Pour les connexions sécurisées.
  - SQLite : Pour la gestion des bases de données.
  - OpenGL ES : Pour le rendu graphique 3D.
- **API d'accès aux services Google :** Permettent d'intégrer les fonctionnalités Google comme les cartes ou la gestion des utilisateurs.

Les Applications et l'Anatomie d'un Déploiement

- Android intègre plusieurs applications par défaut :
  - Navigateur pour internet.
  - Gestionnaire de contacts et application de téléphonie.
- Déploiement d'une application :
  - Les applications Android sont distribuées via des fichiers APK.
  - Le Google Play Store est la principale plateforme de distribution.

La Machine Virtuelle Dalvik et Android Runtime (ART)

- **Dalvik Virtual Machine (DVM) :** Machine virtuelle utilisée avant Android Lollipop.
  - Exécute du bytecode compilé à partir de Java.
- **Android Runtime (ART) :** Remplace **Dalvik** à partir d'Android 5.0 Lollipop.
  - Utilise la compilation anticipée (AOT) pour des performances accrues.

Qu'est-ce que Dalvik ?

- Dalvik est une machine virtuelle open-source utilisée par les systèmes Android.

- Elle exécute des fichiers .dex (*Dalvik Executable*), qui sont plus compacts que les fichiers .class classiques.

- Optimisation de la mémoire :
  - Évite la duplication des chaînes de caractères constantes.
  - Utilise des pointeurs de 32 bits pour adresser les constantes, ce qui réduit l'utilisation de mémoire.

Incompatibilité de Dalvik avec les JVM Classiques

- Dalvik n'est pas compatible avec les JVM classiques telles que Java SE ou Java ME.
- Google a entièrement redéfini les bibliothèques d'accès pour les adapter à Dalvik.

Introduction à ART (Android Runtime)

- Avec Android Lollipop, Android passe de Dalvik à ART (Android Runtime).

- ART utilise la compilation Ahead-of-Time (AOT), ce qui permet de compiler les applications au moment du déploiement et non pendant leur exécution.

Just-in-Time (JIT) vs Ahead-of-Time (AOT)

- **Just-in-Time (JIT) :** La compilation est effectuée pendant l'exécution de l'application.
- **Ahead-of-Time (AOT) :** La compilation est effectuée avant l'exécution, au moment de l'installation de l'application.
- **Avantages de AOT :**
  - Améliore la rapidité d'exécution des applications.
  - Réduit l'utilisation des ressources pendant l'exécution (moins de surcharge).
- **Inconvénients de AOT :**
  - Augmente le temps d'installation des applications, car la compilation se fait à ce moment.

Android Studio (Google)

- **IDE officiel pour Android :** Outil de développement intégré fourni par Google pour créer des applications Android.
- Basé sur *IntelliJ IDEA*, avec des outils spécifiques pour Android.
- Supporte le développement en **Java** et **Kotlin**.
- Inclus des fonctionnalités telles que :
  - Éditeur visuel pour la conception des interfaces.
  - Outils de débogage et de profilage des performances.
  - Émulateur Android intégré pour tester les applications.

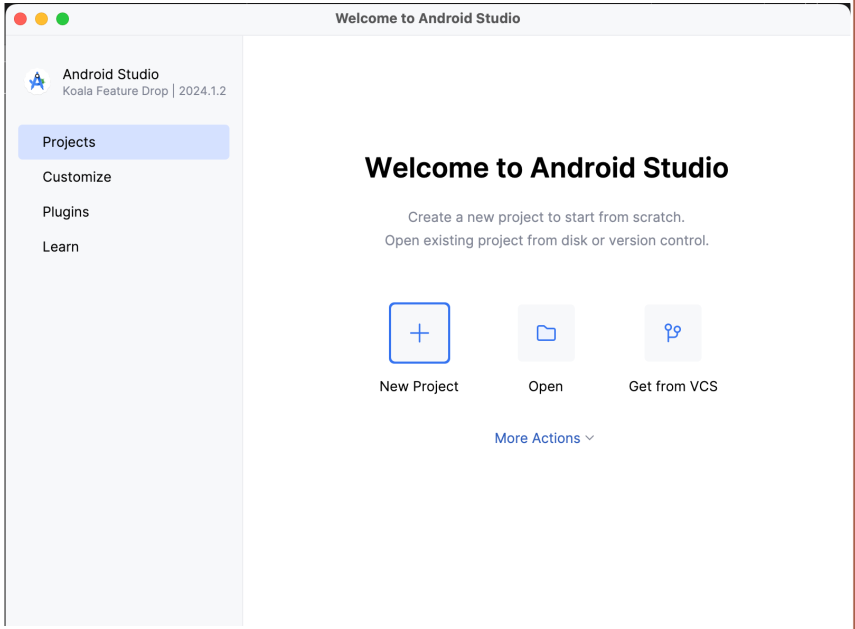
Installation d'Android Studio

- **Étape 1 :** Rendez-vous sur le site officiel de Android Studio : developer.android.com/studio
- **Étape 2 :** Téléchargez la version compatible avec votre système d'exploitation (Windows, macOS, Linux).
- **Étape 3 :** Suivez les instructions d'installation (cliquer sur "Next" dans l'assistant d'installation).
- **Étape 4 :** À la fin de l'installation, lancez Android Studio et configurez le SDK Android.

Un nouveau projet Android

Le nom de l'application et le nom du package

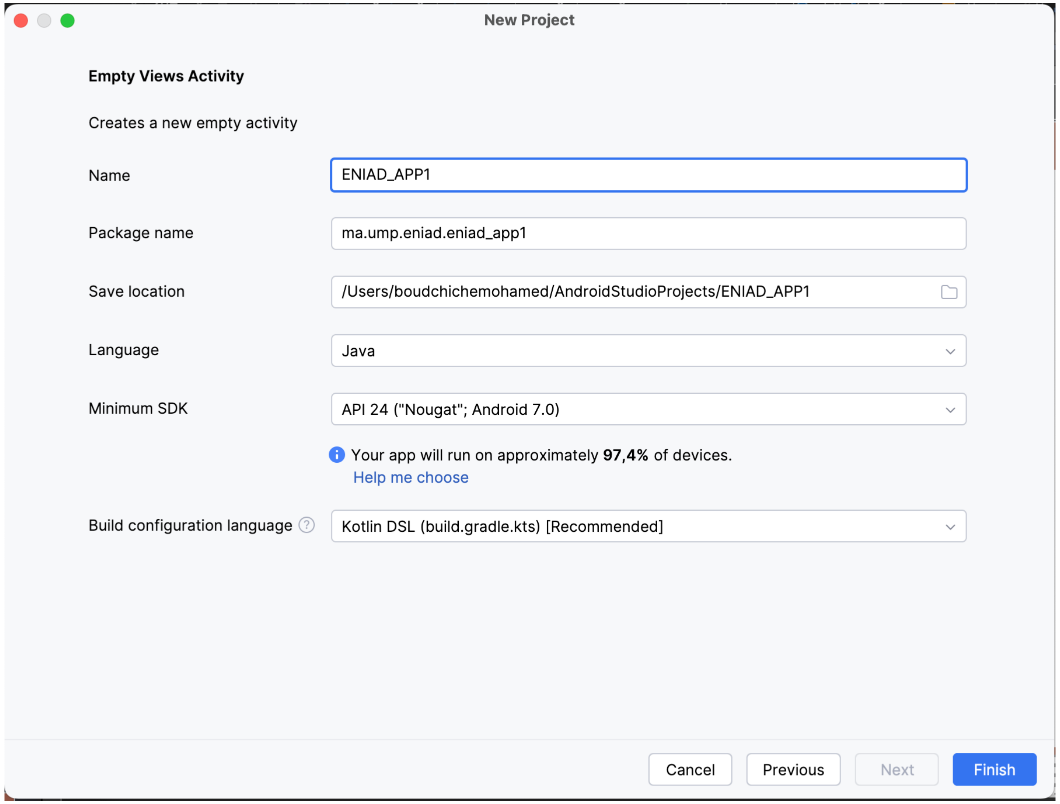
- Pour créer un nouveau projet Android il faut aller au menu suivant :
  - Fichier -> Nouveau -> Nouveau Projet



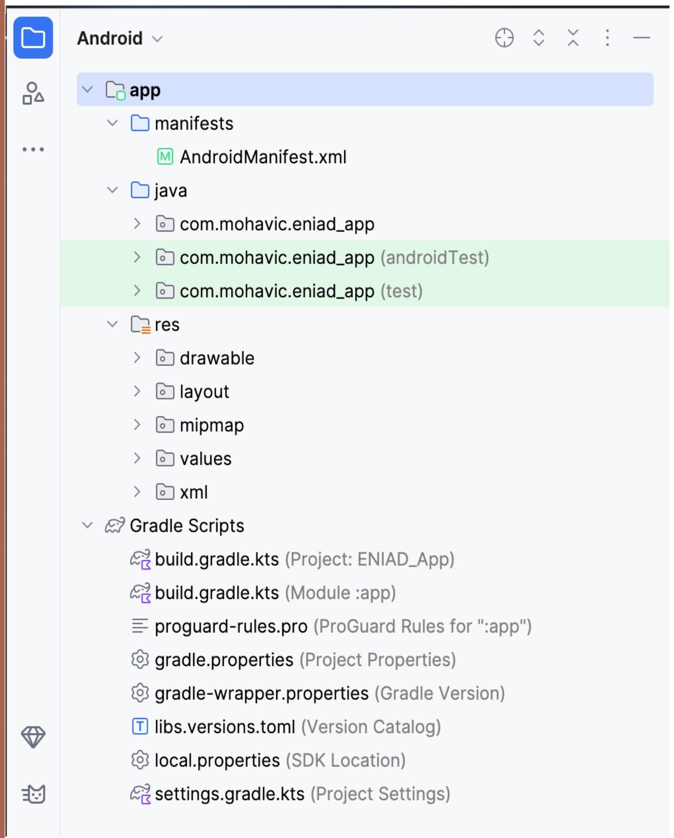
- Le bouton suivant mène à la prochaine étape dans la création du projet Android

Un nouveau projet Android

Le nom de l'application et le nom du package



Structure d'un projet Android Studio



- **Manifests**
  - **AndroidManifest :** le coeur d'une application android. Ce fichier contient la déclaration de toutes les activités et services de l'application ainsi que toutes les permissions requises par l'application.
- **Java**
  - Le code source de l'application: Activités, Services, classes utilitaires…
- **Res**
  - **Drawable :** les images et icones de l'application
  - **Layout :** la définition graphiques des écrans.
  - **Menu :** les menu de l'application
  - **Mipmap :** l'icone (logo) de l'application
  - **Values**
    - <u>Colors :</u> la définition des couleurs à utiliser par l'application
    - <u>Dimens :</u> les dimensions utilisés par l'application
    - <u>Strings :</u> les chaines de caractes (messages) à utiliser dans l'application
- **Gradle :** dependences et parametres de l'application

Les composants d'une Application

- **Les activités (Activity)**
  - Un écran avec une interface utilisateur et un contexte
- **Les services (Service)**
  - Composant sans interface, qui tourne en fond de tâche (lecteur de musique, téléchargement, ...)
- **Les fournisseurs de contenu (ContentProvider)**
  - I/O sur des données gérées par le système ou par une autre application
- **Des récepteurs d'intentions (BroadcastReceiver)**
  - Récupération d'informations générales
  - arrivée d'un sms, batterie faible, …

Les interactions d'une Application

- **Les intentions (Intent)**
  - Permet d'échanger des informations entre composants
  - Démarrage d'un composant en lui envoyant des données
  - Récupération de résultats depuis un composant
  - Recherche d'un composant en fonction d'un type d'action à réaliser
- **Les filtres d'intentions (<intent-filter>)**
  - Permet à un composant d'indiquer ce qu'il sait faire
  - Permet au système de sélectionner les composants susceptibles de répondre à une demande de savoir-faire d'une application

AndroidManifest.xml

- **Description de l'application**
  - ➢ Liste des composants
  - ➢ Niveau minimum de l'API requise
  - ➢ Liste des caractéristiques physiques nécessaires
    - Évite d'installer l'application sur du matériel non compatible (gestion de la visibilité sur Google Play)
  - ➢ Liste des permissions dont l'application a besoin
  - ➢ Liste des autres API nécessaires
    - ex. Google Map
  - ➢ Etc.
- **Généré automatiquement par Android Studio**

Exemple

activity_main.xml   MainActivity.java   AndroidManifest.xml

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

    <application
        android:allowBackup="true"
        android:dataExtractionRules="@xml/data_extraction_rules"
        android:fullBackupContent="@xml/backup_rules"
        android:icon="@mipmap/ic_launcher"
        android:label="ENIAD_App"
        android:roundIcon="@mipmap/ic_launcher_round"
        android:supportsRtl="true"
        android:theme="@style/Theme.ENIAD_App"
        tools:targetApi="31">
        <activity
            android:name=".MainActivity"
            android:exported="true">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />

                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

</manifest>
```

Les ressources

- Définition : Toutes les données autres que le code utilisées par l'application.
- Organisation : Stockées dans le dossier **res** puis incluses dans l'APK.
- Sous-dossiers :
  - **res/drawable** et **res/mipmap** pour les images
  - **Layout** pour les interfaces
  - **Menus** pour les menus
  - **Values** pour les constantes (chaînes, tableaux, valeurs numériques)

Le fichier strings.xml

- Rôle : Contient toutes les chaînes constantes pour l'interface
- Exemples de contenu :

```
<resources>
    <string name="app_name">App_1</string>
</resources>
```

  - Type de la constante : **<string>**
  - Nom de la constante : **app_name**
  - Valeur de la constante : **App_1**
- Utilisation : Facilite la gestion des chaînes de caractères.

Internationalisation

- Objectif :
  - ➢ Disposer de plusieurs versions des textes, libellés, etc utilisés par l'application
  - ➢ Choix automatique des textes en fonction de la configuration du périphérique
- Principe
  - ➢ Dupliquer le fichier strings.xml :

    1 version par langue supportée
  - ➢ Stocker chaque version dans un dossier spécifique
    - values-xx (ex. values-en, values-fr, ...)
  - ➢ Géré via Android Studio

```
app/
  res/
    values/
        strings.xml
    values-en/
        strings.xml
    values-fr/
        strings.xml
```

La classe R

- Classe générée par l'IDE
  - ➢ Permet l'accès aux ressources
  - ➢ Créée à partir de l'arborescence présente dans le dossier res Elle contient des classes internes dont les noms correspondent aux différents types de ressources (drawable, layout, ...)
  - ➢ Elle contient des propriétés permettant de représenter l'ensemble des ressources de l'application
- Utilisation en Java
  - ➢ R. type.identificateur

R.string.app_name

```
<resources>
    <string name="app_name">App_1</string>
    <string name="app_hello">Hello Stu. ENIAD</string>
</resources>
```

R.string.app_hello

Référencement des ressources en XML

- Forme générale : @type/identificateur

@string/app_name

```xml
<resources>
    <string name="app_name">App_1</string>
    <string name="app_hello">Hello Stu. ENIAD</string>
</resources>
```

@string/app_hello

Autres types de ressources

- **Valeur booléenne :**

Syntaxe :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool
        name="bool_name"
        >[true | false]</bool>
</resources>
```

Exemple : Fichier XML enregistré sous **res/values/bools.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <bool name="screen_small">true</bool>
    <bool name="adjust_view_bounds">true</bool>
</resources>
```

Autres types de ressources

- **Couleur :**

Syntaxe :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <color
        name="color_name"
        >hex_color</color>
</resources>
```

Exemple : Fichier XML enregistré sous **res/values/colors.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
   <color name="opaque_red">#f00</color>
   <color name="translucent_red">#80ff0000</color>
</resources>
```

Autres types de ressources

- **ID :**

Syntaxe :

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item
        type="id"
        name="id_name" />
</resources>
```

Exemple : Fichier XML enregistré sous **res/values/ids.xml** :

```xml
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <item type="id" name="button_ok" />
    <item type="id" name="dialog_exit" />
</resources>
```

Autres types de ressources

- **Nombre entier :**

Syntaxe :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer
        name="integer_name"
        >integer</integer>
</resources>
```

Exemple : Fichier XML enregistré sous **res/values/integers.xml** :

```
<?xml version="1.0" encoding="utf-8"?>
<resources>
    <integer name="max_speed">75</integer>
    <integer name="min_speed">5</integer>
</resources>
```

Les activités

- **Définition :** Une activité est un composant d'une application Android offrant une interface utilisateur graphique et un contexte d'exécution.
- **Visibilité :** À tout moment, une seule activité est visible pour l'utilisateur.
    - Utilisation pour la même application ou pour basculer entre différentes applications.
- **Empilement des activités :** Android organise les activités sous forme de pile (backstack).

Cycle de vie d'une activité

- Une activité hérite d'**Activity** directement ou indirectement (**FragmentActivity**, **AppCompatActivity**…)
- Une activité Android passe par plusieurs états durant son cycle de vie:

➢ **onCreate**: permet de gérer les opérations à faire avant l'affichage de l'activité. Dans ce stade l'application n'est toujours visible pour l'utilisateur.

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    Log.d(TAG, "onCreate invoked");    }
```

➢ **onStart**: Cette méthode est invoquée quand l'application devienne visible pour l'utilisateur mais pas encore utilisable (cliquable).

```java
@Override
protected void onStart() {
    super.onStart();
    Log.d(TAG, "onStart invoked");
}
```

➢ **onResume**: l'application est visible et prête pour utilisation.

```java
@Override
protected void onResume() {
    super.onResume();
    Log.d(TAG, "onResume invoked");
}
```

Cycle de vie d'une activité

- **onPause**: une activité peut rester visible mais être mise en pause par le fait qu'une autre activité est en train de démarrer.

```java
@Override
protected void onPause() {
    super.onPause();
    Log.d(TAG, "onPause invoked");
}
```

- **onStop**: l'application n'est plus visible.

```java
@Override
protected void onStop() {
    super.onStop();
    Log.d(TAG, "onStop invoked");
}
```

- **onDestroy**: l'application est détruite complètement.

```java
@Override
protected void onDestroy() {
    super.onDestroy();
    Log.d(TAG, "onDestroy invoked");
}
```

- **onRestart**: cette méthode supplémentaire est appelée quand on relance une activité qui est passée par **onStop**().

```java
@Override
protected void onRestart() {
    super.onRestart();
    Log.d(TAG, "onRestart invoked");
}
```

Puis **onStart**() est aussi appelée. Cela permet de différencier le premier lancement d'un re-lancement.

Sauvegarde des interfaces d'activité

- **Utilisation de *Bundle* dans *onCreate*** : Le Bundle passé en paramètre de *onCreate* permet de restaurer les valeurs de l'interface d'une activité déchargée de la mémoire.

  Exemple : Lorsque l'utilisateur appuie sur le bouton **Home**, Android peut libérer des éléments graphiques de la mémoire. En revenant dans l'application, les valeurs peuvent être restaurées.

- **Limites de la sauvegarde** : Les éléments sans identifiant ne peuvent pas être sauvegardés.

  Si l'application est totalement fermée (tuer l'application), rien n'est restauré.

- **Exemple de code** :

```java
@Override
public void onSaveInstanceState(Bundle outState) {
    outState.putString(SOMEKEY, SOMEVALUE);
    super.onSaveInstanceState(outState);
}
```

Packaging et déploiement

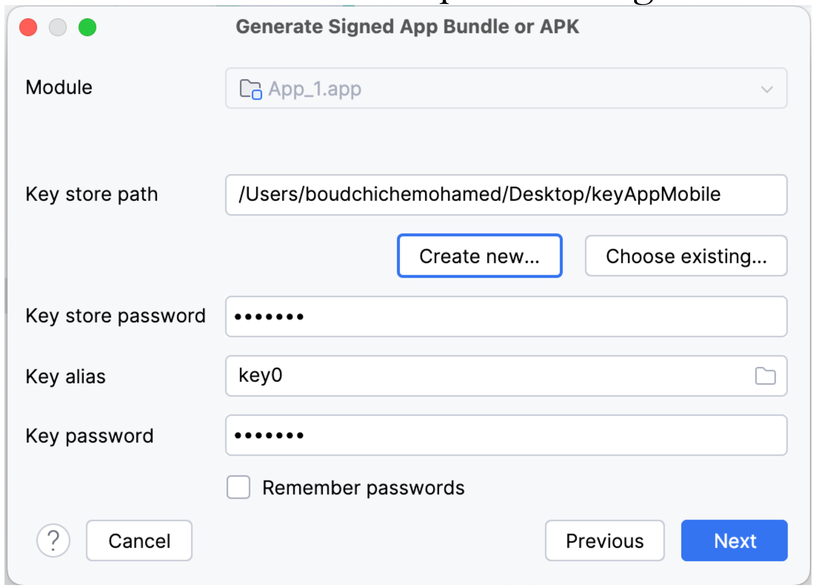
- Une application android est un fichier avec une extension .apk
- Une application pourrait être installé soit en double cliquant sur le fichier apk ou bien en téléchargeant l'application depuis les store.
- Il y a deux types de store:
  - ➢ Store Public → PlayStore
  - ➢ Store Entreprise → Appalooza, Fabric,…

Le fichier apk doit être signé avec un certificat avant la soumission vers les stores. Pour cela, il faut générer une application signé

Packaging et déploiement

Générer une Application signé

- Pour générer une application signée, il faut aller vers le menu: Build → Generate Signed APK
- Apres avoir choisi l'application à signer, la fenêtre suivante s'affichera pour renseigner les paramètres de signature.



- La première option permet d'indiquer le chemin du certificat ou bien de créer un nouveau
- Les trois options: **keystore passowrd**, **key alias**, **key password** permettent de paramétrer le certificat crée
- En cliquant sur suivant, l'assistant demande de choisir le chemin où va générer l'apk signé

03

Conception d'Interfaces Graphiques Mobiles.

Quelques règles de base

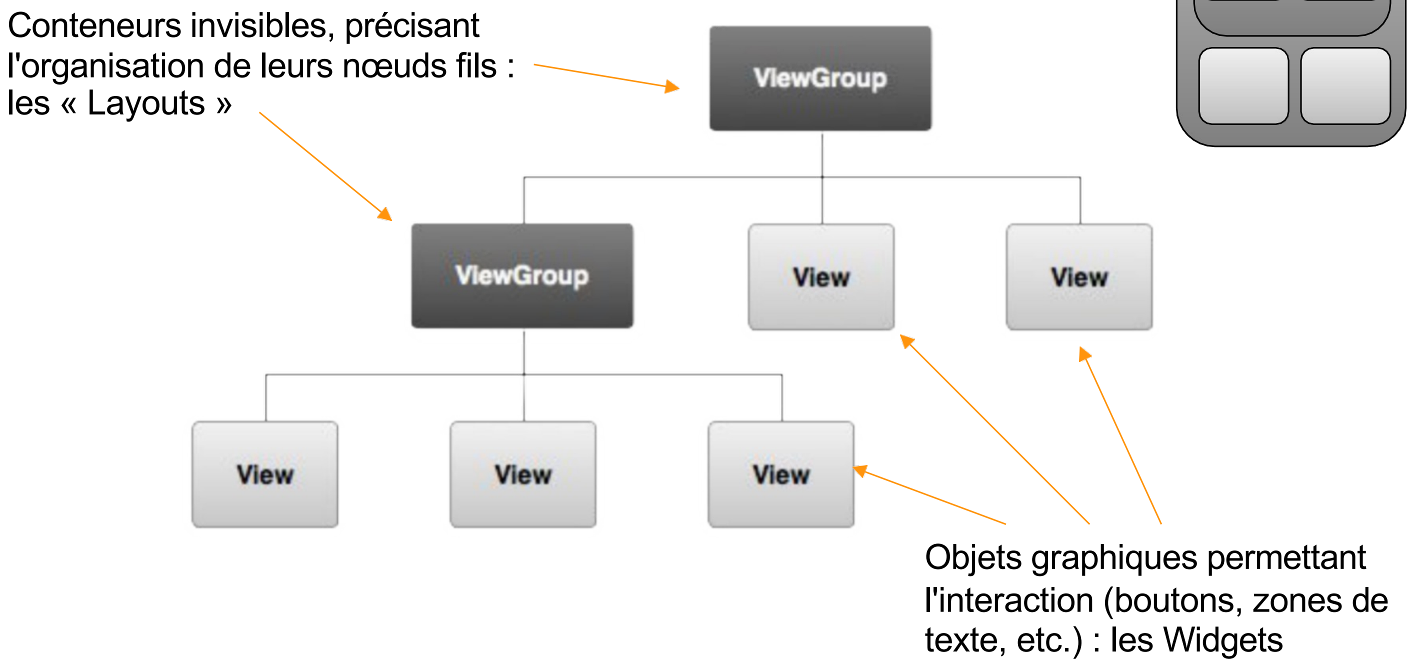
- Interface = **seul** contact de l'utilisateur
  - Faire attirant
  - Faire simple
    - L'application doit être intuitive
    - Éviter les trop longs messages
- Faire ergonomique
  - L'enchaînement des activités doit être rapide
  - L'utilisateur doit toujours connaître l'état courant de l'activité
- Conseils et « matériels » :
  - http://developer.android.com/design

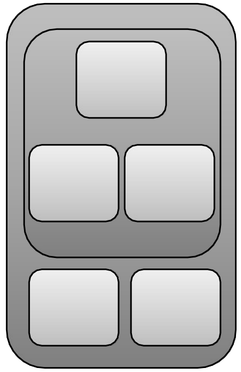
Définir une interface graphique

- Définir les « interacteurs »
  - Objets graphiques visibles par l'utilisateur pour :
    - L'affichage (texte, images, etc.)
    - L'interaction (boutons, cases, champs de saisie, etc.)
- Définir leur mise en page
  - Positions dans l'interface (fixes ou relatives)
- XML ou Java (sauf traitement de l'interaction : Java seul)
    - Privilégier XML
      - Souplesse de mise à jour
      - Permet la prise en compte simplifiée de différents types d'écran

Représentation d'une interface

- Représentation arborescente





Les Layouts

- Zone invisible assurant l'organisation automatique des composants graphiques
  - Peuvent être déclarées en XML ou Java
  - Privilégier XML
    - Séparation du code et de la mise en page
    - Souplesse d'adaptation à différents périphériques
- Possèdent des propriétés « intuitives » permettant l'organisation des composants
- Nombreux *layouts* différents
  - Peuvent être imbriqués (cf arborescence)
- Un *layout* doit être chargé dans onCreate()
  - setContentView(R.layout.nom_du_layout)

Les Layouts

- Gestion multi-écrans
  - Différentes tailles
    - small, normal, large, xlarge
  - Différentes densités de pixels
    - ldpi (120dpi), mdpi (160 dpi), hdpi (240 dpi), xhdpi (320 dpi), xxhdpi (480 dpi), xxxhdpi (640 dpi)
  - Prévoir un *layout* par taille (et orientation) de l'écran si nécessaire
    - effets de positionnements relatifs pouvant être gênants
  - Prévoir des images en différentes résolutions

Les Layouts

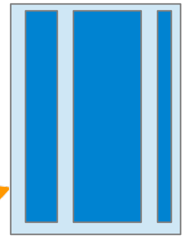
- Fonctionnement similaire à l'internationalisation
  - Un sous-dossier spécifique à chaque *layout* et/ou à chaque image

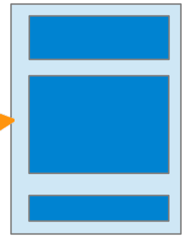
```
MyProject/
    res/
        layout/              # default (portrait)
            main.xml
        layout-land/         # landscape
            main.xml
        layout-large/        # large (portrait)
            main.xml
        layout-large-land/   # large landscape
            main.xml

MyProject/
    res/
        drawable-xhdpi/
            awesomeimage.png
        drawable-hdpi/
            awesomeimage.png
        drawable-mdpi/
            awesomeimage.png
        drawable-ldpi/
            awesomeimage.png
```

Source : developer.android.com

LinearLayout

- Aligne les nœuds dans une seule direction
  - horizontale (par défaut)
  - verticale





activity_main.xml MainActivity.java

```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">

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

    <TextView
        android:id="@+id/textView"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Coucou" />

</LinearLayout>
```

Taille du composant :

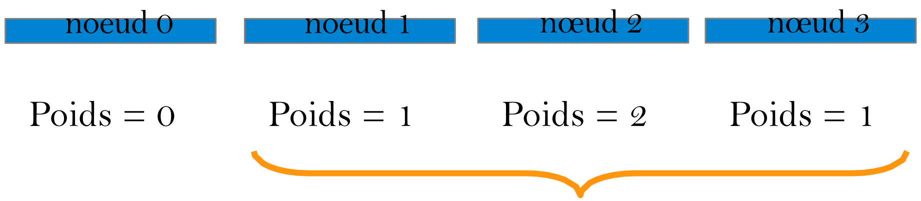
match_parent : S'adapte à la taille du conteneur parent (ici l'écran)

wrap_content : s'adapte à la taille de ce qu'il contient (ici une zone de texte)

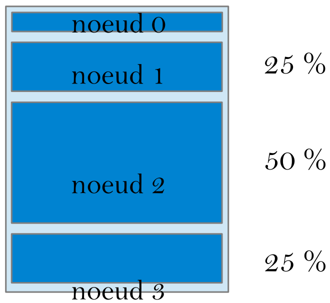
dimension fixe

LinearLayout

- Modification du « poids » de chaque nœud
  - Permet de changer la taille de la zone occupée par chaque nœud dans l'écran
  - Ajout d'un attribut **android:layout_weight** à chaque nœud
    - 0 (par défaut) : n'utilise que la zone nécessaire au nœud
    - n>0 : poids du nœud par rapport aux autres nœuds



Répartition au prorata du poids



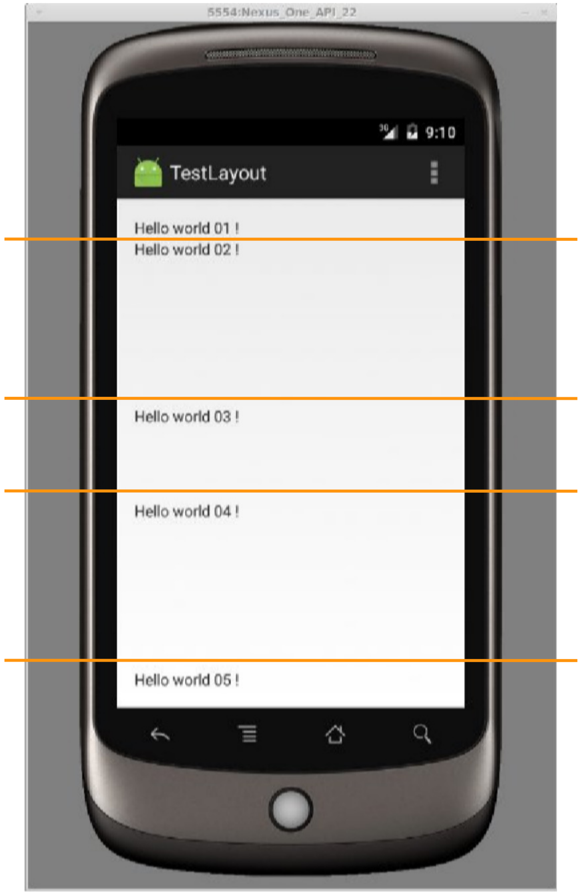
LinearLayout

```xml
<TextView
    android:text="@string/hello_world_01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />

<TextView
    android:text="@string/hello_world_02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="2"
    />
<TextView
    android:text="@string/hello_world_03"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="1"
    />
<TextView
    android:text="@string/hello_world_04"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_weight="2"
    />
<TextView
    android:text="@string/hello_world_05"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
```

40 %

20 %

40 %



LinearLayout

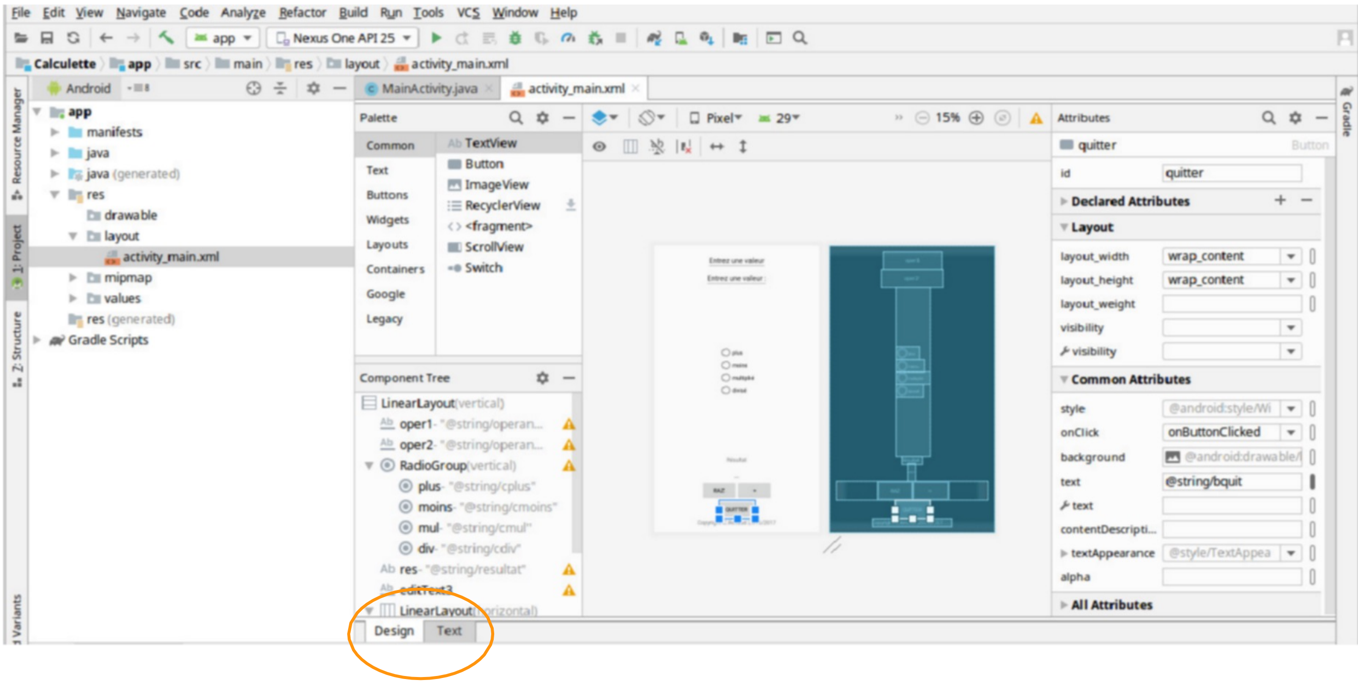
- Alignement de chaque noeud dans sa zone
  - Ajout d'un attribut **android:layout_gravity**
  - Nombreuses valeurs possibles :
    - center, center_vertical, center_horizontal
    - left, right, top, bottom
    - Etc. (cf LinearLayout.LayoutParams)

LinearLayout

```
<TextView
    android:text="@string/hello_world_01"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />

<TextView
    android:text="@string/hello_world_02"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center_horizontal"
    android:layout_weight="2"
    />
<TextView
    android:text="@string/hello_world_03"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="right"
    android:layout_weight="1"
    />
<TextView
    android:text="@string/hello_world_04"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:gravity="center"
    android:layout_weight="2"
    />
<TextView
    android:text="@string/hello_world_05"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_gravity="right"
    />
```

Remarque

- Possibilité d'organiser visuellement les layouts sous Android Studio



ConstraintLayout

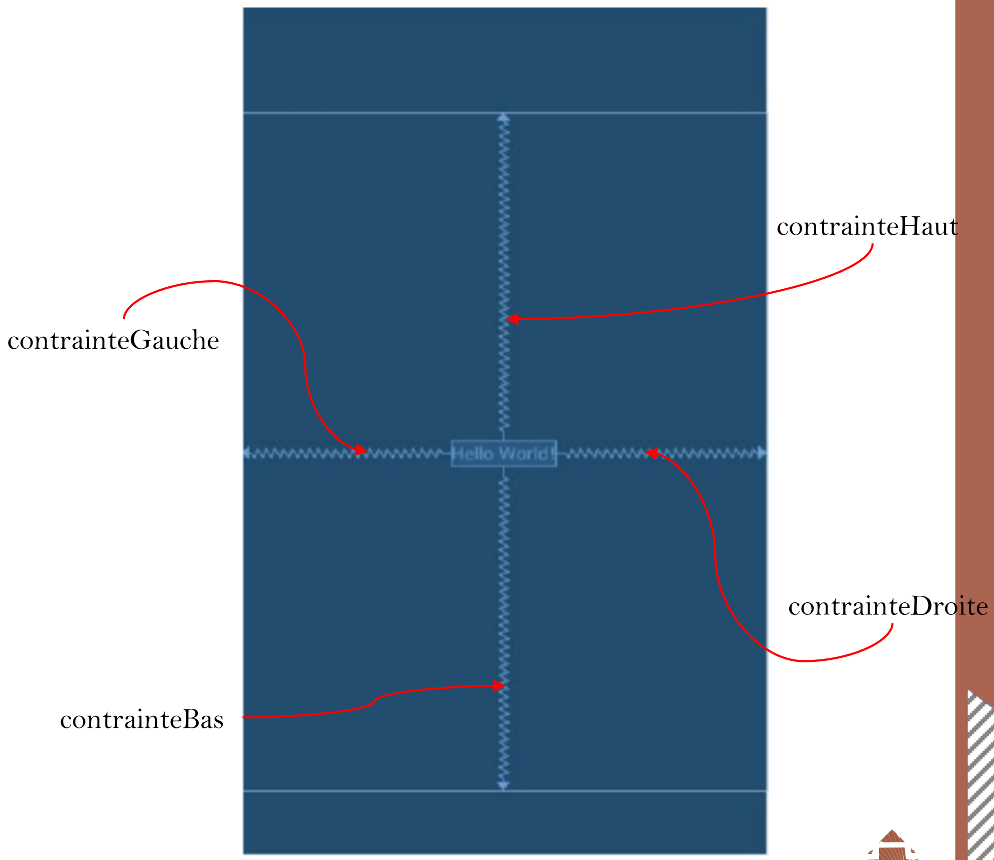
- Objectif :
  - Faciliter la création de *layouts* complexes
  - Réduire l'imbrication de *layouts*
    - Consommatrice de ressources
- Organisation intuitive :
  - Création de relations entre les composants graphiques et avec leur *layout* parent
    - Notion de contraintes
  - Très flexible
  - Bien adapté à l'éditeur graphique

ConstraintLayout

- Contraintes
  - Au moins une contrainte horizontale
  - Au moins une contrainte verticale

Exemple :

```
<android.support.constraint.ConstraintLayout
xmlns:android=
   "http://schemas.android.com/apk/res/android"
   xmlns:app="http://schemas.android.com/apk/res-auto"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   tools:context=".MainActivity">

    <TextView
       android:id="@+id/textView"
       android:layout_width="wrap_content"
       android:layout_height="wrap_content"
       android:text="Hello World!"
       app:layout_constraintBottom_toBottomOf="parent"
       app:layout_constraintLeft_toLeftOf="parent"
       app:layout_constraintRight_toRightOf="parent"
       app:layout_constraintTop_toTopOf="parent"  />

</android.support.constraint.ConstraintLayout>
```

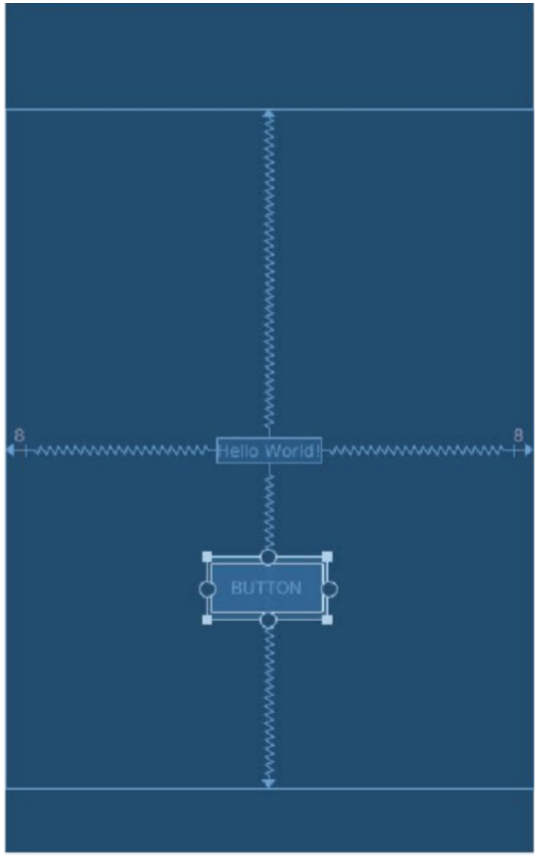


ConstraintLayout

Remarque

- Placement sans contrainte d'un composant graphique via l'éditeur graphique
  - Apparaît au bon endroit dans l'éditeur
- Placé par défaut en (0,0) lors de l'exécution …

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        tools:layout_editor_absoluteX="148dp"
        tools:layout_editor_absoluteY="373dp" />
```

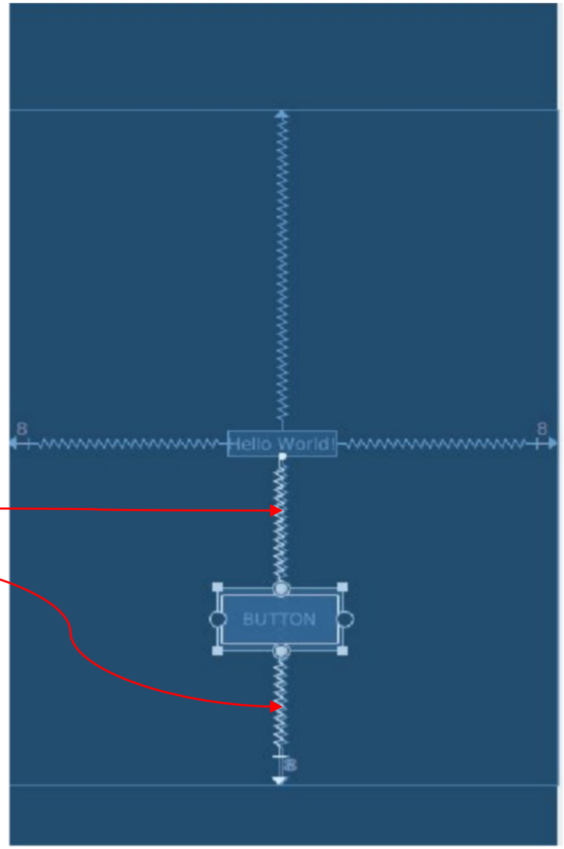


ConstraintLayout

- Ajout d'une contrainte verticale

```
<Button
    android:id="@+id/button"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginTop="8dp"
    android:text="Button"
    app:layout_constraintTop_toBottomOf="@+id/textView "
    app:layout_constraintBottom_toBottomOf="parent"
    tools:layout_editor_absoluteX="148dp" />
```

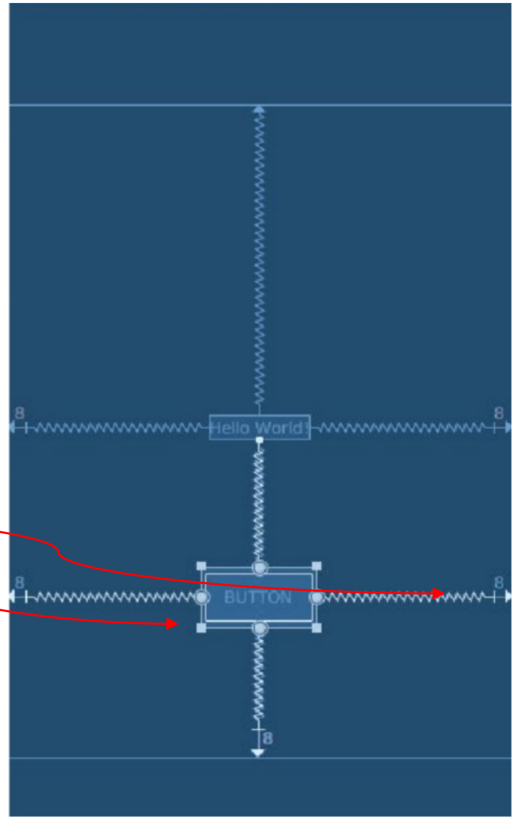
X=0



ConstraintLayout

- Ajout d’une contrainte horizontale

```
<Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_marginBottom="8dp"
        android:layout_marginEnd="8dp"
        android:layout_marginLeft="8dp"
        android:layout_marginRight="8dp"
        android:layout_marginStart="8dp"
        android:layout_marginTop="8dp" android:text="Button"
        app:layout_constraintBottom_toBottomOf="parent"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintStart_toStartOf="parent"
        app:layout_constraintTop_toBottomOf="@+id/textView"  />
```

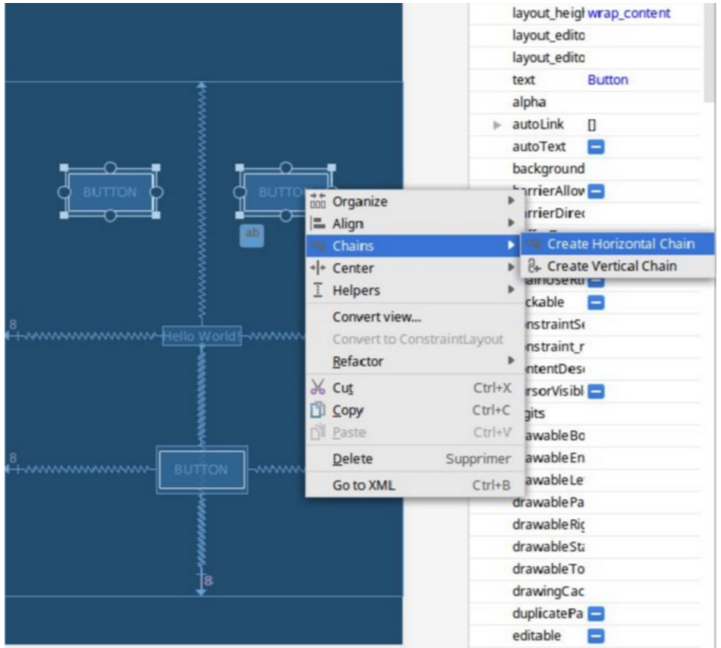


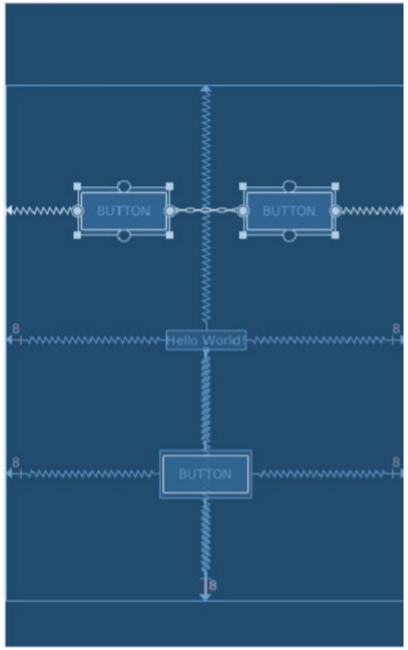
Remarque :

- `@+id/nom` rajoute nom à la classe R uniquement si `nom` n’existe pas
- Utile si le composant nommé est défini après dans le fichier xml

ConstraintLayout

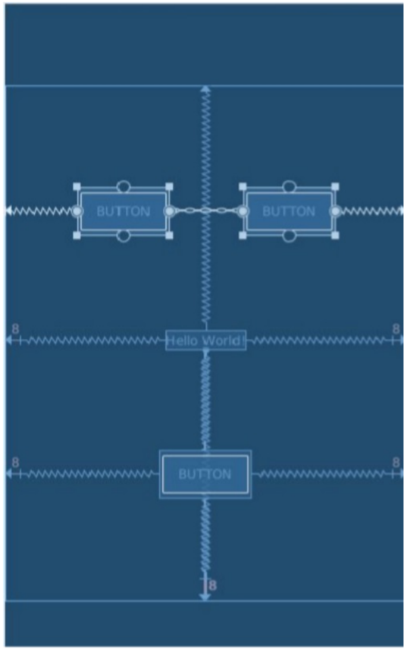
- Création de chaînages





Pas de contraintes verticales (y = 0)

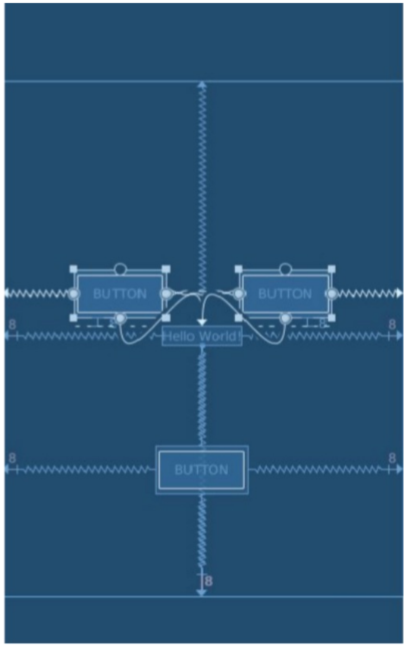
ConstraintLayout

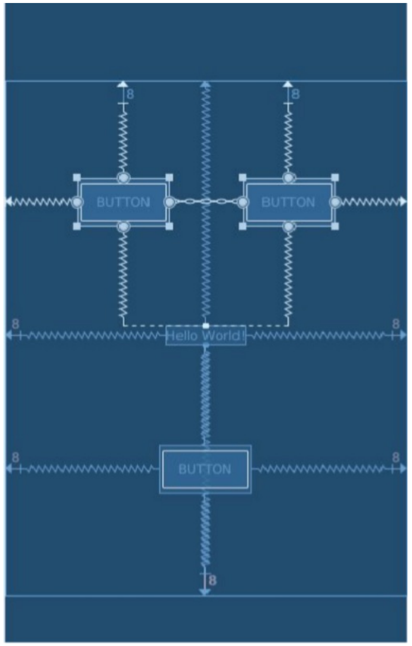



```
<Button
        android:id="@+id/button3"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintEnd_toStartOf="@+id/button4"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toStartOf="parent"
        tools:layout_editor_absoluteY="101dp" />

    <Button
        android:id="@+id/button4"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Button"
        app:layout_constraintEnd_toEndOf="parent"
        app:layout_constraintHorizontal_bias="0.5"
        app:layout_constraintStart_toEndOf="@+id/button3"
        tools:layout_editor_absoluteY="101dp" />
```

ConstraintLayout

app:layout_constraintBottom_toTopOf="@+id/textView "





app:layout_constraintTop_toTopOf="parent"

Les Widgets

- Composants graphiques visibles par l'utilisateur
  - *Widgets* simples : zones de texte, boutons, listes, etc.
  - *Widgets* plus complexes : horloges, barres de progression, etc.
- Héritent de la classe **View**
- Utilisation :
  - Définition en XML (type, taille, centrage, position, etc.)
  - Comportement en Java
  - Peuvent également être créés dynamiquement en Java

Les TextView

- *Widget* permettant l'affichage d'un texte
  - Normalement non éditable
- Exemple :

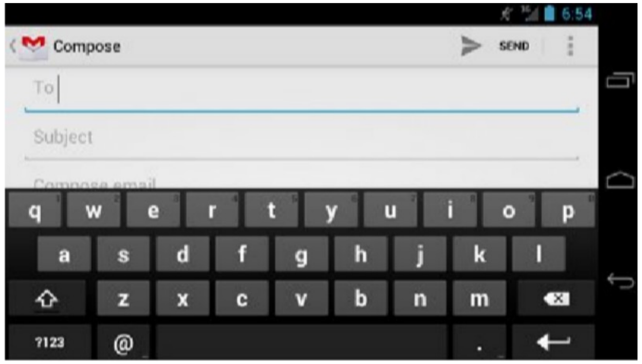
```
<TextView
        android:id="@+id/text"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/letexte"
        android:hint="texte initial"
        android:layout_gravity="center"
        android:gravity="center" />
```

- Nombreux autres attributs
  - Cf classe TextView

Les EditText

- *Widget* permettant la saisie d'un texte (TextFields)
  - Accès : ouverture d'un clavier pour la saisie
  - nombreux attributs permettant l'aide à la saisie

```
<EditText
    android:id="@+id/email_address"
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:hint="@string/email_hint"
    android:inputType="textEmailAddress" />
```



"text" : Normal text keyboard.
"textEmailAddress" : Normal text keyboard with the @ character.
"textUri" : Normal text keyboard with the / character.
"number" : Basic number keypad.
"phone" : Phone-style keypad.

Source : developer.android.com

Les Button

- *Widget* représentant un bouton d'action
  - Renvoie un événement lors de l'appui
  - Peut contenir un texte, une image ou les deux
- Exemples :

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    ... />
```

```
<Button
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@string/button_text"
    android:drawableLeft="@drawable/button_icon"
    ... />
```

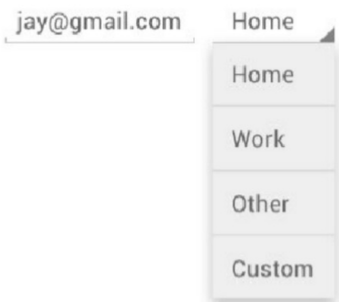

```
<ImageButton
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:src="@drawable/button_icon"
    ... />
```

Source : developer.android.com

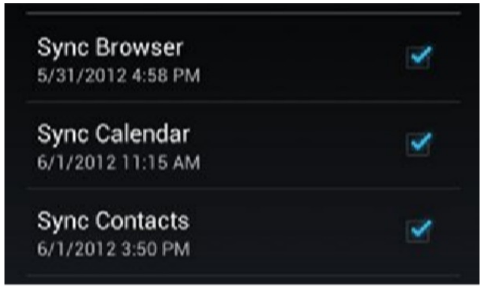
80

En vrac …

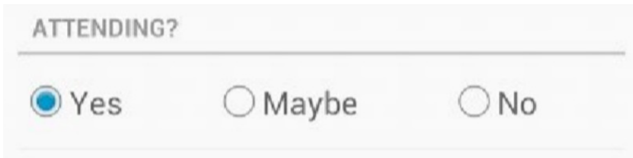
- Quelques autres widgets
  - Source developer.android.org



Spinner



CheckBox



RadioButton

ToggleButton

Switch
(android 4.0+)

Implantation du comportement

- Les fichiers XML ne permettent que de :
  - positionner les composants ;
  - définir leurs caractéristiques.
- Nécessité de :
  - définir leur comportement
    - type d'interaction (clic court, clic long, etc.)
    - code de prise en compte (Java)
  - lier composant et code (XML ou Java)
    - XML : attribut android:onClick
    - Java : instancier un *event listener*

Implantation du comportement

- Attribut android:onClick
  - Doit être suivi du nom de la méthode à appeler en cas de déclenchement
  - Prototype :
    - public void nomDeLaMethode(View maVue)

```
<Button
        android:id="@+id/monBouton"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/monTexte"
        android:onClick="onBoutonClique "   />
```

```
public void onBoutonClique(View maVue) {
        System.out.println("le bouton a été cliqué") ;
}
```

Permet de récupérer des informations sur le composant graphique qui a généré l'événement
Initialisé par le système avant l'appel

Récupération :

maVue.getId() ➡ R.id.monBouton

Implantation du comportement

- Les *event listener*
  - interfaces de la classe View
  - ne disposent que d'une seule méthode à implanter
  - méthode appelée quand le composant associé est déclenché par l'utilisateur
- Exemples :

Interface	Méthode
View.OnClickListener	abstract void onClick(View v)
View.OnLongClickListener	abstract boolean onLongClick(View v)
View.OnFocusChangeListener	abstract void onFocusChange(View v, boolean hasFocus)

Implantation du comportement

- Exemple : l'interface View.onClickListener
  - public void onClick(View v)

Retrouver un widget à partir du nom qui lui a été associé dans le xml

```
...
Button button = (Button) findViewById(R.id.button_name);

button.setOnClickListener(new View.OnClickListener()   {

    public void onClick(View v) {
        // Do something in response to button click
    }
});
...
```

Source : developer.android.com

Création d'un « OnClickListener »

Surcharge de la méthode « onClick » de l'interface « onClickListener »

Associer un « onClickListener » au bouton

Exercices

1. Quel attribut XML est utilisé pour centrer un élément à la fois verticalement et horizontalement dans un `LinearLayout` ?
   a) ‘android:gravity’
   b) ‘android:layout_gravity’
   c) 'android:align_center’
   d) ‘android:center’
2. Quel type de layout est recommandé pour gérer des interfaces complexes et réduire l'imbrication de layouts ?
   a) FrameLayout
   b) RelativeLayout
   c) ConstraintLayout
   d) TableLayout

Exercices

1. Quelle est la meilleure pratique pour déclarer des interfaces dans Android ?
    a) Toujours utiliser Java pour toute l'interface
    b) Utiliser XML pour la mise en page et Java pour les interactions
    c) Utiliser Java pour la mise en page et XML pour les interactions
    d) Toujours utiliser uniquement XML
2. Quelle est l'utilité de l'attribut `android:onClick` dans un fichier XML de layout ?
    a) Permet d'indiquer la méthode qui sera appelée lors d'un clic sur le composant
    b) Définit l'alignement d'un composant par rapport à son parent
    c) Spécifie la couleur du composant lorsqu'il est cliqué
    d) Affecte l'effet de gravité sur un composant

Les Intents : Définition et Utilisation

- Un **Intent** est un objet de communication qui permet de demander une action d'un autre composant d'application.
- Les Intents permettent plusieurs types d'actions, notamment :
  - **Démarrer une activité :**
    - Une activité représente un écran unique dans une application. Pour lancer une nouvelle instance, utilisez la méthode startActivity().
    - L'Intent décrit l'activité à démarrer et transmet les données nécessaires.
    - **Exemple :** Si vous souhaitez recevoir un résultat à la fin de cette activité, utilisez startActivityForResult.

Utilisation des Intents : Services et Broadcasts

1. **Démarrer un Service :**
   - Les services permettent d'exécuter des opérations en arrière-plan, sans interface utilisateur.
   - Utilisez startService() pour démarrer un service.
   - Depuis Android 5.0 (API 21), vous pouvez utiliser **JobScheduler** pour gérer les tâches d'arrière-plan de manière optimisée.
2. **Envoyer un Broadcast :**
   - Un **broadcast** est un message que toute application peut recevoir.
   - Utilisez sendBroadcast() pour transmettre un message vers d'autres applications.
   - **Exemples d'utilisation :** GPS, GMAIL, YouTube, et autres fonctionnalités système qui nécessitent des Intents pour communiquer.

Les Intents : Types d'Intents

- **INTENT Explicite :**
  - Un Intent explicite spécifie le composant exact à exécuter en utilisant le **nom de classe pleinement qualifié**.
  - Ce type d'Intent est souvent utilisé pour démarrer un composant au sein de sa propre application, lorsque le nom de la classe de l'activité ou du service est connu.
  - **Exemple d'utilisation :** Démarrer une nouvelle activité en réponse à une action de l'utilisateur ou lancer un service pour télécharger un fichier en arrière-plan.

Les Intents : Types d'Intents

- **INTENT Implicite :**
  - Un Intent implicite ne spécifie pas un composant précis. Au lieu de cela, il déclare une **action générale** à accomplir, permettant à d'autres applications compatibles de gérer cette action.
  - **Exemple d'utilisation :** Si vous souhaitez afficher un emplacement sur une carte, vous pouvez utiliser un Intent implicite pour demander qu'une application compatible affiche l'emplacement sur une carte.

Les Intents : Ajouter des Informations (Extras)

- **Les Extras dans les Intents :**
  - Les Intents permettent de transporter des informations supplémentaires appelées **Extras** vers l'activité cible.
  - Les méthodes utilisées pour ajouter et récupérer ces informations sont putExtra et getExtra.
  - Chaque information est ajoutée sous forme de **clé -> valeur**.
- **Exemple d'ajout d'Extras :**

```
Intent callActivityAuthentication = new Intent(getApplicationContext(),
AuthenticationActivity.class);
callActivityAuthentication.putExtra("login", "yog");
callActivityAuthentication.putExtra("nom", "yog");
callActivityAuthentication.putExtra("prenom", "yog");
callActivityAuthentication.putExtra("age", 25);
startActivity(callActivityAuthentication);
```

Les Intents : Récupérer les Informations (Extras)

- **Récupérer les Extras dans l'activité cible :**
  - Pour accéder aux informations envoyées par l'Intent, on utilise getIntent().getExtras() pour obtenir les valeurs.
  - Exemple de récupération des valeurs :

```
Bundle extras = getIntent().getExtras();
String login = new String(extras.getString("login"));
String nom = new String(extras.getString("nom"));
String prenom = new String(extras.getString("prenom"));
Integer age = new Integer(extras.getInt("age"));
```

Remarque Importante :

- Si vous devez transmettre un objet complexe via un Intent, cet objet doit implémenter l'interface **Serializable** ou **Parcelable**.

Les Intents : Types d'Actions

- **Définition des Actions dans les Intents :**
  - Le type d'action est le premier paramètre dans la construction d'un Intent.
  - Les actions peuvent être **natives** (fournies par le système) ou **définies par le développeur**.
- **Actions Natives :**
  - Android fournit plusieurs actions par défaut, par exemple Intent.ACTION_VIEW, qui permet d'ouvrir une application pour visualiser un contenu via une URI.

Exemple d'Utilisation :

- Utilisation de l'action Intent.ACTION_SEND pour envoyer un email, nécessitant des informations supplémentaires ajoutées avec putExtra.

Exemples d'Actions et Actions Personnalisées

➢ **Exemple d'Envoi d'Email :**

```
Intent emailIntent = new Intent(android.content.Intent.ACTION_SEND);
String[] recipients = new String[]{"email@email.com", ""};
emailIntent.putExtra(android.content.Intent.EXTRA_EMAIL, recipients);
emailIntent.putExtra(android.content.Intent.EXTRA_SUBJECT, "Subject");
emailIntent.putExtra(android.content.Intent.EXTRA_TEXT, "Message");
emailIntent.setType("text/plain");
startActivity(Intent.createChooser(emailIntent, "Send mail..."));
```

➢ **Création d'une Action Personnalisée :**

- Pour définir une action unique, il suffit de créer un Intent et de spécifier l'action.

```
Intent monIntent = new Intent("intent");
monIntent.setAction("action_à_communiquer");
```

Les Intents : Catégories

- **Définition des Catégories d'Intents :**
  - Les catégories permettent de regrouper les applications en fonction de grands types de fonctionnalités, comme les clients d'email, les navigateurs, les lecteurs de musique, etc.
- **Exemples de Catégories :**
  - Ces catégories facilitent le lancement d'applications spécifiques selon leur type de fonctionnalité.

Détails des Catégories d'Intents

- **DEFAULT** : Catégorie par défaut utilisée si aucune autre catégorie spécifique n'est mentionnée.
- **BROWSABLE** : Permet de lancer une activité depuis un lien dans un navigateur, facilitant ainsi la gestion de nouveaux types de liens (e.g., foo://truc).
- **APP_MARKET** : Catégorie pour ouvrir le marché des applications et télécharger des applications.
- **APP_MUSIC** : Utilisée pour lancer des activités de lecture et de navigation de musique.

Les Intents : Lancer une Nouvelle Activité Interne

- **Création d'un Intent :**
  - Il existe plusieurs constructeurs pour créer un objet de type **Intent** afin de lancer une nouvelle activité.
  - Lorsqu'il s'agit d'une activité interne à l'application, on peut créer l'Intent en passant le **contexte** et la classe de l'activité cible.
- **Exemple de Code :**

```java
Intent loginIntent = new Intent(getApplicationContext(), AuthenticationActivity.class);
startActivity(loginIntent);
```

Remarque sur le Contexte :

- Le premier paramètre de l'Intent est le **contexte** de l'application. Utilisez getApplicationContext() lorsque l'objet qui manipule l'Intent n'hérite pas de Context.

Les Intents : Lancer une Activité Externe avec URI

- **Lancer une Autre Application :**
  - Pour transmettre la main à une autre application, on utilise un Intent avec l'**URI cible** et le type d'action (comme Intent.ACTION_DIAL pour passer un appel).
  - Le système se charge de trouver une application compatible pour exécuter l'action spécifiée.
- **Exemple de Code :**

```
Button buttonCall = (Button) findViewById(R.id.button);
buttonCall.setOnClickListener(new View.OnClickListener() {
    public void onClick(View v) {
        Uri telnumber = Uri.parse("tel:0600000000");
        Intent call = new Intent(Intent.ACTION_DIAL, telnumber);
        startActivity(call);
    }
});
```

Les Intents : Retour d'une Activité

➢ **Démarrer une activité avec retour de résultat :**

- Une activité peut être démarrée avec startActivityForResult() pour obtenir un résultat, au lieu de startActivity().
- Exemple : Lancer l'application Contacts pour que l'utilisateur sélectionne un contact, et recevoir les coordonnées en retour.

➢ **Exemple de Code :**

```
Intent pickContactIntent = new Intent(Intent.ACTION_PICK, Uri.parse("content://contacts"));
startActivityForResult(pickContactIntent, PICK_CONTACT_REQUEST);
```

Explication :

- PICK_CONTACT_REQUEST est un code entier utilisé pour identifier la requête et savoir quel type de résultat est attendu.

Les Intents : Traitement du Résultat

- **Recevoir et Filtrer les Résultats :**
  - Dans la classe parente, la méthode onActivityResult() est utilisée pour savoir si l'activité enfant a retourné un résultat.
  - Cette méthode permet de vérifier le code de requête, le code de résultat, et de lire les données retournées.
- **Exemple de Code pour le Retour de Données :**

```
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == PICK_CONTACT_REQUEST) {
        if (resultCode == RESULT_OK) {
            // Le contact sélectionné est renvoyé ici
            // Utiliser l'URI pour obtenir les détails du contact
        }
    }
}
```

Envoi d'un Résultat à la Classe Parente :

```
Intent returnIntent = new Intent();
returnIntent.putExtra("result", result);
setResult(Activity.RESULT_OK, returnIntent);
finish();
```

Les Intents : Broadcasting des Informations

- **Broadcasting avec les Intents :**
  - Il est possible d'utiliser un **Intent** pour diffuser un message à but informatif.
  - Toutes les applications abonnées à cette action peuvent recevoir et capturer le message diffusé.
- **Exemple de Code :**

```
Intent monIntent = new Intent("intent");
monIntent.setAction("action_à_communiquer");
monIntent.putExtra("extra_key", "extra_value");
sendBroadcast(monIntent);
```

Explication :

- Le nom de l'action (action_à_communiquer) doit être unique pour identifier clairement le broadcast.

Utilisation des Broadcasts dans Android

- **Réception de Messages de Broadcast :**
  - Les applications Android peuvent envoyer et recevoir des messages de broadcast venant d'autres applications ou du système.
- **Exemples de Diffusions Système :**
  - Android envoie automatiquement des broadcasts pour divers événements système, comme le passage en mode avion.
  - Les applications abonnées sont automatiquement notifiées de ces événements.

Remarque :

- Le nom de l'action doit être unique pour différencier chaque broadcast et éviter les interférences avec d'autres messages.

Les Intents : Recevoir et Filtrer les Intents

- **Nécessité des Filtres :**
  - Avec la multitude de messages véhiculés par les Intents, chaque application doit pouvoir écouter uniquement les Intents dont elle a besoin.
  - Android permet d'utiliser des **filtres déclaratifs** dans le Manifest pour spécifier quels types d'Intents l'application peut gérer.
- **Niveaux de Filtrage :**
  - **Action** : Identifie le nom de l'Intent. Utilisez une convention de nommage pour éviter les collisions.
  - **Catégorie** : Permet de filtrer par catégorie d'action (par exemple, DEFAULT, BROWSABLE).
  - **Data** : Filtre les données, par exemple en utilisant android:host pour limiter à un domaine spécifique.

Déclaration des Filtres d'Intents dans le Manifest

- **Déclaration d'un Filtre :**
  - Dans le fichier Manifest, en ajoutant un filtre au niveau du tag <activity>, une application peut déclarer les types de messages qu'elle sait gérer.
- **Exemple de Code :**

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

Explication :

- Cet exemple montre comment spécifier une action et une catégorie dans le filtre pour restreindre les Intents reçus par l'activité.

Les Intents : Réception par un BroadcastReceiver (Configuration Statique)

- **Réception des Messages de Broadcast :**
  - Les messages broadcastés peuvent être captés par une classe héritant de **BroadcastReceiver**.
  - Cette classe doit surcharger la méthode onReceive pour traiter le message reçu.
- **Déclaration dans le Manifest :**
  - Pour que le BroadcastReceiver soit actif, il doit être défini dans le fichier **AndroidManifest.xml**.
  - Le tag <receiver> dans le Manifest permet de pointer vers la classe chargée de recevoir les messages.
- **Exemple de Code dans le Manifest :**

```
<application android:icon="@drawable/icon" android:label="@string/app_name">
    <receiver android:name="MyRedefinedBroadcastReceiver">
        <intent-filter>
            <action android:name="action_à_communiquer" />
            <category android:name="android.intent.category.DEFAULT" />
        </intent-filter>
    </receiver>
</application>
```

Implémentation de la Classe BroadcastReceiver

- **Création de la Classe Receiver :**
  - La classe doit étendre **BroadcastReceiver** et surcharger la méthode onReceive.
- **Exemple de Code de la Classe BroadcastReceiver :**

```
public class MyRedefinedBroadcastReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Bundle extra = intent.getExtras();
        if (extra != null) {
            String val = extra.getString("extra");
            Toast.makeText(context, "Broadcast message received: " + val, Toast.LENGTH_SHORT).show();
        }
    }
}
```

Explication :

- Dans cet exemple, le message reçu est affiché sous forme de **Toast** avec la valeur extraite du bundle.

Les Intents : Réception par un BroadcastReceiver Dynamique

- **Récepteur Dynamique :**
  - Il est possible de créer un **BroadcastReceiver** de manière dynamique en appelant la méthode registerReceiver.
  - Il faut fournir un objet receiver et un **IntentFilter** avec le nom de l'action à recevoir.
- **Exemple de Code :**

```
MyRedefinedBroadcastReceiver myreceiver = new MyRedefinedBroadcastReceiver();
IntentFilter filtre = new IntentFilter("action_à_communiquer");
registerReceiver(myreceiver, filtre);
```

Explication :

- Cette méthode est pratique pour enregistrer un receiver uniquement lorsque l'application ou une activité spécifique est active.

Désenregistrement et Utilisation Interne du BroadcastReceiver

- **Désenregistrement du Receiver :**
  - Une fois le receiver inutile, il est essentiel de le **désenregistrer** pour libérer les ressources.
  - Utilisez unregisterReceiver(myreceiver); pour éviter les fuites de mémoire.
- **Déclaration Interne :**
  - Il est possible de définir un **BroadcastReceiver** directement dans l'activité pour implémenter onReceive sans créer une classe séparée.
- **Exemple de Déclaration Interne :**

```java
BroadcastReceiver broadcast = new BroadcastReceiver() {
    @Override
    public void onReceive(Context context, Intent intent) {
        // Code pour gérer le message reçu
    }
};
```

Les Intents : Messages Natifs de l'OS

- **Messages Broadcast de l'OS :**
  - Android émet un certain nombre de messages natifs permettant de surveiller divers événements système.
- **Exemples de Messages :**
  - **ACTION_BOOT_COMPLETED** : Diffusé lorsque le système a terminé son démarrage.
  - **ACTION_SHUTDOWN** : Diffusé lorsque le système est en cours d'extinction.
  - **ACTION_SCREEN_ON / OFF** : Indique l'allumage ou l'extinction de l'écran.
  - **ACTION_POWER_CONNECTED / DISCONNECTED** : Indique la connexion ou la déconnexion de l'alimentation.
  - **ACTION_TIME_TICK** : Notification envoyée chaque minute.
  - **ACTION_USER_PRESENT** : Notification reçue lorsque l'utilisateur déverrouille son téléphone.

Actions de Broadcast pour Lancer des Applications

- **Documentation des Broadcasts :**
  - Tous les messages de broadcast disponibles se trouvent dans la documentation officielle des **Intents**.
- **Autres Actions pour Déléguer un Traitement :**
  - **ACTION_CALL (ANSWER, DIAL)** : Passer, réceptionner ou afficher un appel téléphonique.
  - **ACTION_SEND** : Envoyer des données via SMS ou e-mail.
  - **ACTION_WEB_SEARCH** : Effectuer une recherche sur internet.

Barre d’action et menus

Barre d'action

La barre d'action contient l'icône d'application (1), quelques items de menu (2) et un bouton . pour avoir les autres menus (3).

Réalisation d'un menu

Le principe général : un menu est une liste de d'items présentés dans la barre d'action. La sélection d'un item déclenche une *callback*.

Docs Android sur la barre d'action et sur les menus Il faut définir :

- un fichier res/menu/*nom_du_menu*.xml qui est une sorte de layout spécialisé pour les menus,
- deux méthodes d'écouteur pour gérer les menus :
    - ajout du menu dans la barre,
    - activation de l'un des items.

Spécification d'un menu

Créer res/menu/*nom_du_menu*.xml :

```
<menu xmlns:android="..." >
    <item android:id="@+id/menu_creer"
        android:icon="@drawable/ic_menu_creer"
        android:showAsAction="ifRoom"
        android:title="@string/menu_creer"/>
    <item  android:id="@+id/menu_chercher"  ...  />
    ...
</menu>
```

L'attribut showAsAction vaut "always", "ifRoom" ou "never" selon la visibilité qu'on souhaite dans la barre d'action. Cet attribut est à modifier en app:showAsAction si on utilise *androidx*.

Icônes pour les menus

Android distribue gratuitement un grand jeu d'icônes pour les menus, dans les deux styles *MaterialDesign* : HoloDark et HoloLight.

Téléchargez l'Action Bar Icon Pack pour des icônes à mettre dans vos applications.

Écouteur pour afficher le menu

Il faut programmer deux méthodes. L'une affiche le menu, l'autre réagit quand l'utilisateur sélectionne un item. Voici la première :

```
@Override
public boolean onCreateOptionsMenu(Menu menu)
{
    // ajouter mes items de menu
    getMenuInflater().inflate(R.menu.nom_du_menu,     menu);
    // ajouter les items du système s'il y en a
    return super.onCreateOptionsMenu(menu);
}
```

Cette méthode rajoute les items du menu défini dans le XML.

Un `MenuInflater` est un lecteur/traducteur de fichier XML en vues ; sa méthode `inflate` crée les vues.

Écouteur pour afficher le menu

On peut aussi ajouter des éléments de menu manuellement :

```
public static final int MENU_ITEM1 = 1;
...

@Override
public boolean onCreateOptionsMenu(Menu menu)
{
  ...
  // ajouter des items manuellement
  menu.add(Menu.NONE, MENU_ITEM1, ordre, titre).setIcon(image);
  ...
}
```

Cette fois, vous devrez choisir un identifiant pour les items. L'ordre indique la priorité de cet item ; mettre Menu.NONE s'il n'y en a pas.

Réactions aux sélections d'items

Voici la seconde *callback*, c'est un aiguillage selon l'item choisi :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
    case R.id.menu_creer:
        ...
        return true;
    case MENU_ITEM1:
        ...
        return true;
    ...
    default: return super.onOptionsItemSelected(item);
    }
}
```

Réactions aux sélections d'items

Mais, dans les versions récentes d'Android Studio, les identifiants de menu ne sont plus des constantes et ne peuvent plus être utilisés dans un switch. On doit le transformer en conditionnelles en cascade :

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    int id = item.getItemId();
    if (id == R.id.menu_creer) {
        ...
        return true;
    } else if (id == MENU_ITEM1) {
        ...
        return true;
    ...
    } else return super.onOptionsItemSelected(item);
}
```

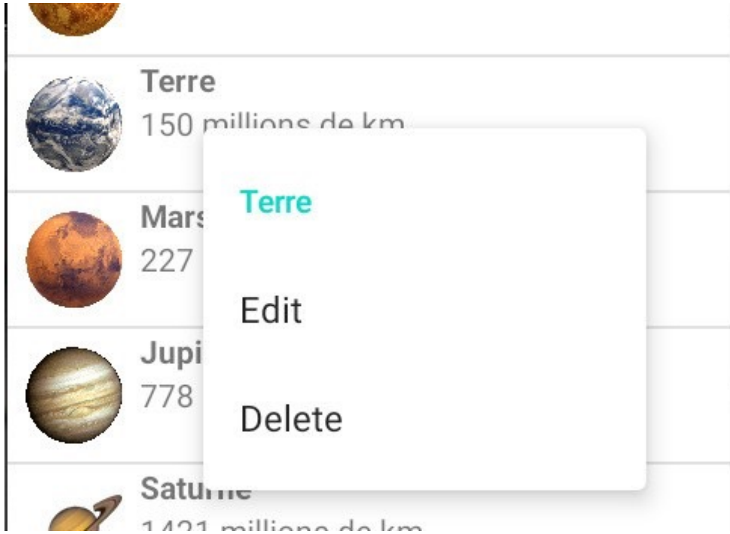
Menus en cascade

Définir deux niveaux quand la barre d’action est trop petite :

```
<menu xmlns:android="..." >
    <item android:id="@+id/menu_item1" ... />
    <item android:id="@+id/menu_item2" ... />
    <item android:id="@+id/menu_more"
        android:icon="@drawable/ic_action_overflow"
        android:showAsAction="always"
        android:title="@string/menu_more">
        <menu>
            <item android:id="@+id/menu_item3" ... />
            <item android:id="@+id/menu_item4" ... />
        </menu>
    </item>
</menu>
```

Menus contextuels

Menus contextuels



Ces menus apparaissent généralement lors un clic long sur un élément de liste. La classe RecyclerView ne possède rien pour afficher automatiquement des menus. Il faut soi-même programmer le nécessaire.

Menus contextuels

Voici les étapes :

- Le *View Holder* doit se déclarer en tant qu'écouteur pour des ouvertures de menu contextuel (clic long).
- La méthode déclenchée fait apparaître le menu contextuel. L'activité doit se déclarer en tant qu'écouteur pour les clics sur les éléments du menu contextuel.

Le souci principal, c'est qu'il n'y a pas de lien entre d'une part le *View Holder* qui observe le clic long sur un élément de la liste et fait afficher le menu contextuel, et d'autre part l'activité qui est réveillée quand l'utilisateur sélectionne un item du menu.

Il faut fournir la position de l'élément de la liste à l'activité, et on est obligé de bricoler.

View Holder écouteur de menu

Il suffit d'ajouter ceci :

```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
{
    private PlaneteBinding ui;

    public PlaneteViewHolder(@NonNull PlaneteBinding ui)
    {
        super(ui.getRoot());
        this.ui = ui;
        // pour faire apparaître le menu contextuel
        itemView.setOnCreateContextMenuListener(
                this::onCreateContextMenu);
    }
```

itemView est une variable de la classe ViewHolder qui est égale à `ui.getRoot()`.

View Holder écouteur de menu

Voici la *callback* d'un clic long sur un élément de la liste :

```
public static final int MENU_EDIT = 1;
public static final int MENU_DELETE = 2;

private void onCreateContextMenu(ContextMenu menu, View v,
        ContextMenu.ContextMenuInfo   menuInfo)
{
    // position de l'élément
    int position = getAdapterPosition();
    // stocker la position dans l'ordre (3e paramètre)
    menu.add(Menu.NONE, MENU_EDIT, position, "Edit");
    menu.add(Menu.NONE, MENU_DELETE, position, "Delete");
    // titre du menu
    menu.setHeaderTitle(ui.nom.getText());
}
```

View Holder écouteur de menu

Il y a une astuce, mais un peu faible : utiliser la propriété `order` des items de menu pour stocker la position de l'élément cliqué dans la liste. Cette propriété permet normalement de les classer pour les afficher dans un certain ordre, mais comme on ne s'en sert pas. . .

On est obligé de faire ainsi car il manque une propriété *custom* dans la classe MenuItem. Une meilleure solution serait de sous-classer cette classe avec la propriété qui nous manque, mais il faudrait modifier beaucoup plus de choses.

Une dernière remarque : il n'est pas possible d'afficher un icône à côté du titre d'item. C'est un choix délibéré dans Android.

Écouteur dans l'activité

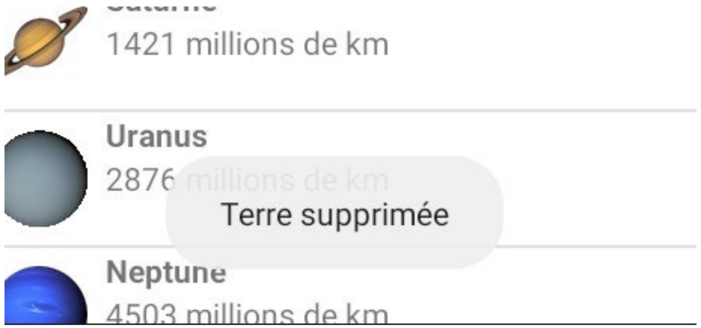
Enfin, il reste à rendre l'activité capable de recevoir les événements d'un clic sur un item du menu :

```
@Override
public boolean onContextItemSelected(MenuItem item) {
    int position = item.getOrder();           // récup position
    Planete planete = liste.get(position);    // récup item
    switch (item.getItemId()) {
      case PlaneteViewHolder.MENU_EDIT:
          // TODO éditer planete (activité ou fragment...)
          return true;
      case PlaneteViewHolder.MENU_DELETE:
          // TODO supprimer planete (dialogue confirmation...)
          return true;
    }
    return false;    // menu inconnu, non traité ici
}
```

Annonces et dialogues

Annonces : toasts

Un « *toast* » est un message apparaissant en bas d'écran pendant un instant, par exemple pour confirmer la réalisation d'une action. Un *toast* n'affiche aucun bouton et n'est pas actif.



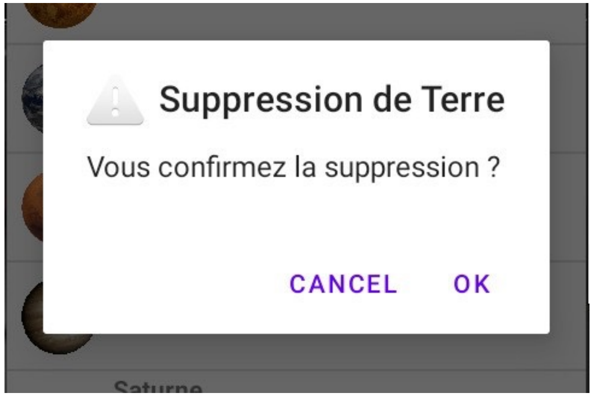
Voici comment l'afficher avec une ressource chaîne :

```
Toast.makeText(getContext(),
    R.string.item_supprime,  Toast.LENGTH_SHORT).show();
```

La durée d'affichage peut être allongée avec LENGTH_LONG.

Dialogues

Un dialogue est une petite fenêtre qui apparaît au dessus d'un écran pour afficher ou demander quelque chose d'urgent à l'utilisateur, par exemple une confirmation.



Il existe plusieurs sortes de dialogues :

- Dialogues d'alerte
- Dialogues généraux

Dialogue d’alerte

Un dialogue d’alerte AlertDialog affiche un texte et un à trois boutons au choix : ok, annuler, oui, non, aide. . .

Un dialogue d’alerte est construit à l’aide d’une classe nommée AlertDialog.Builder. Le principe est de créer un *builder* et c’est lui qui crée le dialogue. Voici le début :

```
AlertDialog.Builder builder = new AlertDialog.Builder(this);
builder.setTitle("Suppression de "+planete.getNom());
builder.setIcon(android.R.drawable.ic_dialog_alert);
builder.setMessage("Vous confirmez la suppression ?");
```

Ensuite, on rajoute les boutons et leurs écouteurs.

NB: utiliser des ressources pour les chaînes.

Boutons et affichage d'un dialogue d'alerte

Le *builder* permet de rajouter toutes sortes de boutons : oui/non, ok/annuler. . . Cela se fait avec des fonctions comme celle-ci. On peut associer un écouteur (anonyme privé ou . . .) ou aucun.

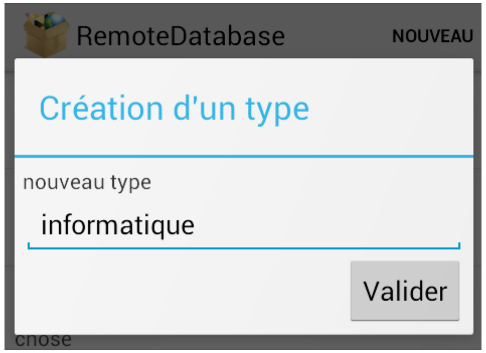
```
// rajouter un bouton "oui" qui supprime vraiment
builder.setPositiveButton(android.R.string.ok,
    // écouteur écrit sous la forme d'une lambda
    (dialog, idbtn) -> supprimerVraiment(planete));
// rajouter un bouton "non" qui ne fait rien
builder.setNegativeButton(android.R.string.cancel, null);
```

Enfin, on affiche le dialogue :

```
// affichage du dialogue
builder.show();
```

Dialogues personnalisés

Lorsqu'il faut demander une information plus complexe à l'utilisateur, mais sans que ça nécessite une activité à part entière, il faut faire appel à un dialogue personnalisé.



Création d'un dialogue

Il faut définir le layout du dialogue incluant tous les textes, sauf le titre, et au moins un bouton pour valider, mais pas pour annuler car on peut fermer le dialogue avec le bouton back.

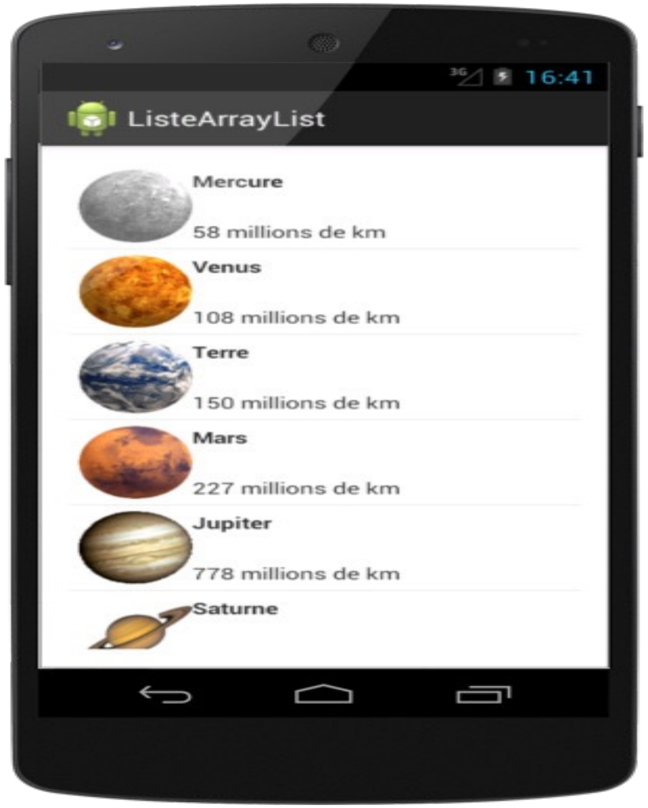
```xml
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="..." ...>
    <TextView android:id="@+id/titre" .../>
    <EditText android:id="@+id/libelle" .../>
    <Button android:id="@+id/valider" ... />
</LinearLayout>
```

Ensuite cela ressemble à ce qu'on fait dans onCreate d'une activité : installation du layout et des écouteurs pour les boutons.

Affichage du dialogue

```
Dialog dialog = new Dialog(this);
DialogPersoBinding dialogUI =
            DialogPersoBinding.inflate(getLayoutInflater());
dialog.setContentView(dialogUI.getRoot());
dialog.setTitle("Création d'un type");
// bouton valider
dialogUI.valider.setOnClickListener(v -> {
        // récupérer et traiter les infos
        ...
        // ne surtout pas oublier de fermer le dialogue
        dialog.dismiss();
    }
});
// afficher le dialogue
dialog.show();
```

Présentation d'une liste d'items


16:41
ListeArrayList
Mercure
58 millions de km
Venus
108 millions de km
Terre
150 millions de km
Mars
227 millions de km
Jupiter
778 millions de km
Saturne

Principe général

On veut programmer une application pour afficher et éditer une liste d'items.

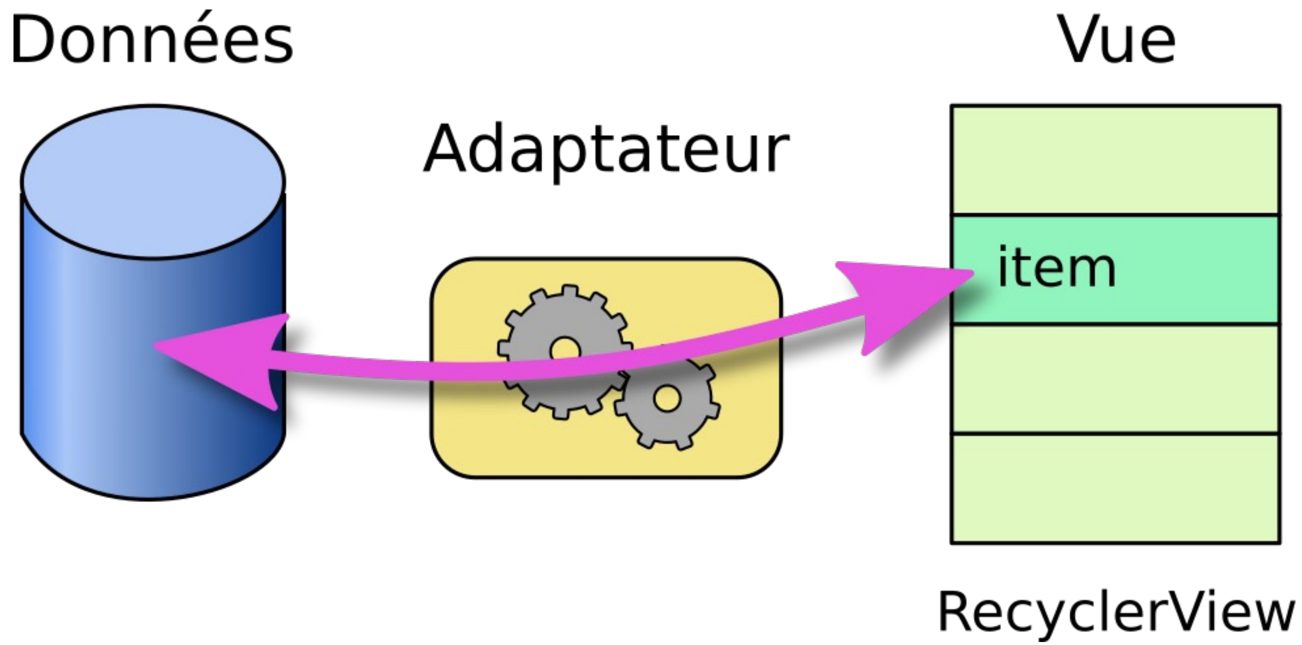
Cette semaine, la liste est stockée dans un tableau type `ArrayList` ; après, ça sera dans une BDD *Realm*.

L'écran est occupé par un RecyclerView. C'est une vue spécialisée dans l'affichage de listes quelconques.

Anciennement, on utilisait des `ListView`, mais ils sont délaissés car trop peu polyvalents.

Schéma global

Modèle MVC : le contrôleur entre les données et la vue s'appelle un *adaptateur*.



Une classe pour représenter les items

Pour commencer, il faut représenter les données :

```
public class Planete {
    public String nom; // nom de la planète
    public int distance; // distance au soleil

    Planete(String nom, int distance) {
        this.nom = nom;
        this.distance = distance;
    }
}
```

Lui rajouter tous les accesseurs (*getters*) et modificateurs (*setters*) pour en faire un *JavaBean* : objet Java simple (POJO) composé de variables membres privées initialisées par le constructeur, et d'accesseurs.

Données initiales

Deux solutions pour initialiser la liste avec des valeurs prédéfinies :

Un tableau dans les ressources.

Un tableau constant Java comme ceci :

```
final Planete[] initdata = {
    new Planete("Mercure", 58),
    new Planete("Vénus", 108),
    new Planete("Terre", 150),
    ...
};
```

`final` signifie constant, `initdata` ne pourra pas être réaffecté (par contre, ses cases peuvent être réaffectées).

Copie dans un ArrayList

L'étape suivante consiste à recopier les valeurs initiales dans un tableau dynamique de type ArrayList<Planete> :

```
private List<Planete> liste;

void onCreate(...)
{
    ...

    // copie du tableau initdata dans le ArrayList
    liste = new ArrayList<>(Arrays.asList(initdata));
}
```

NB: `Arrays.asList` crée une liste non modifiable, c'est pour ça qu'on la recopie dans un `ArrayList`.

Rappels sur le container List<type>

C'est un type de données *générique*, c'est à dire paramétré par le type des éléments mis entre `<...>` ; ce type doit être un objet.

```
List<TYPE> liste = new ArrayList<>();
```

NB: le type entre <> à droite est facultatif.

La variable est du type `List` (superclasse abstraite) et affectée avec un `ArrayList`. La raison est qu'il faut de préférence toujours employer le type le plus général qui possède les méthodes voulues. Mais quand c'est une classe abstraite (une interface), on l'instancie avec une sous-classe non-abstraite.

Par exemple un `List` peut être instancié avec un `ArrayList` ou un `LinkedList`. On choisit en fonction des performances voulues : un `ArrayList` est très rapide en accès direct, mais très lent en insertion. C'est l'inverse pour un `LinkedList`.

Rappels sur le container List<type>, suite

Quelques méthodes utiles de la classe abstraite `List`, héritées par `ArrayList` :

- `liste.size()` : retourne le nombre d'éléments présents,
- `liste.clear()` : supprime tous les éléments,
- `liste.add(elem)` : ajoute cet élément à la liste,
- `liste.remove(elem ou indice)` : retire cet élément
- `liste.get(indice)` : retourne l'élément présent à cet indice,
- `liste.contains(elem)` : `true` si elle contient cet élément,
- `liste.indexOf(elem)` : indice de l'élément, s'il y est.

Données initiales dans les ressources

On crée deux tableaux dans le fichier res/values/arrays.xml

```
<resources>
    <string-array  name="noms">
        <item>Mercure</item>
        <item>Venus</item>
        ...
    </string-array>
    <integer-array   name="distances">
        <item>58</item>
        <item>108</item>
        ...
    </integer-array>
</resources>
```

Intérêt : traduire les noms des planètes dans d'autres langues en créant des variantes, ex: res/values-en/arrays.xml

Données dans les ressources

Ensuite, on récupère ces ressources tableaux pour remplir le `ArrayList` :

```
// accès aux ressources

Resources res = getResources();
final String[] noms = res.getStringArray(R.array.noms);
final int[] distances = res.getIntArray(R.array.distances);

// recopie dans le ArrayList
liste = new ArrayList<>();
for (int i=0; i<noms.length; ++i) {
    liste.add(new Planete(noms[i], distances[i]));
}
```

C’est plus complexe, mais préférable à la solution du tableau pré-initialisé, pour bien séparer programme et données.

Affichage de la liste

Activité

L’affichage de la liste est fait par un RecyclerView. C’est une vue qui intègre un défilement automatique et qui veille à économiser la mémoire pour l’affichage.

Voici le layout le plus simple qui remplit tout l’écran, mais on peut rajouter d’autres vues : boutons. . . :

```
<androidx.recyclerview.widget.RecyclerView
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:id="@+id/recycler"   android:layout_width="match_parent"
    android:layout_height="match_parent"      />
```

Notez la présence du package de cette vue dans la balise. Elle fait partie de l’ensemble androidx.

Mise en place du layout d'activité

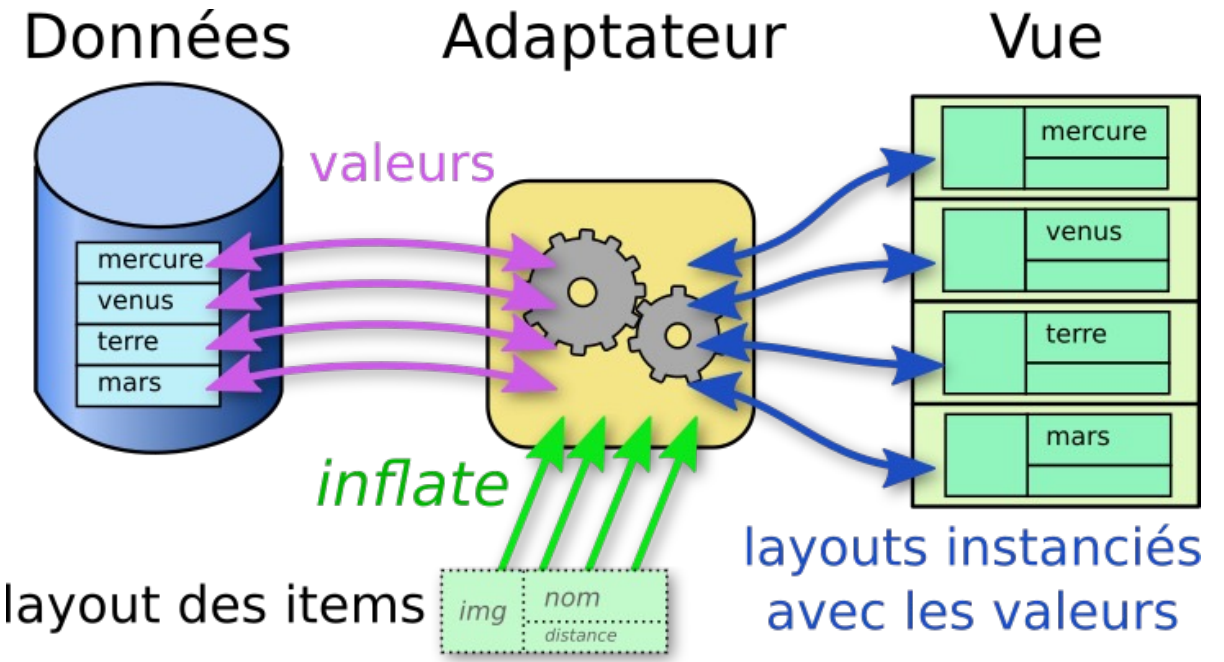
En utilisant un ViewBinding :

```
private  ArrayList<Planete>  liste;
private ActivityMainBinding ui;

@Override protected void onCreate(Bundle savedInstanceState)
{
    // mettre en place le layout contenant le RecyclerView
    super.onCreate(savedInstanceState);
    ui = ActivityMainBinding.inflate(getLayoutInflater());
    setContentView(ui.getRoot());

    // initialisation de la liste avec les ressources
    liste = new ArrayList<>();
    ...
```

Adaptateurs et ViewHolders

Relations entre la vue et les données

Un RecyclerView affiche les items à l'aide d'un *adaptateur* :



Concepts

La vue ne sert qu'à afficher les éléments de la liste. En réalité, seuls quelques éléments seront visibles en même temps. Cela dépend de la hauteur de la liste et la hauteur des éléments.

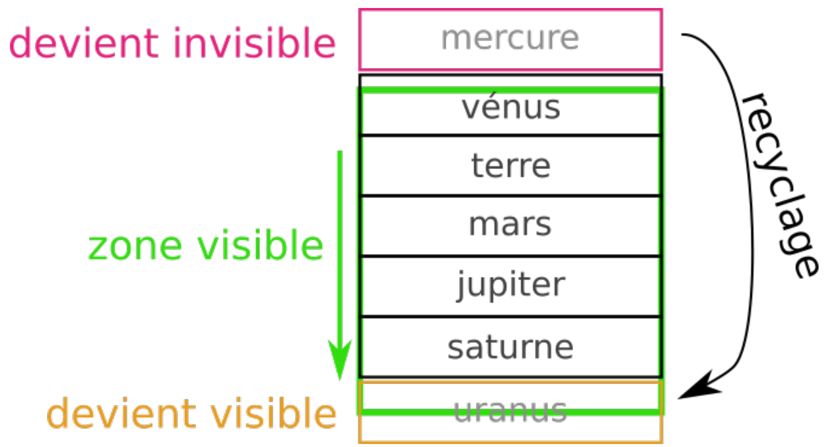
Le principe du RecyclerView est de ne gérer que les éléments visibles. Ceux qui ne sont pas visibles ne sont pas mémorisés. Mais lorsqu'on fait défiler la liste ainsi qu'au début, de nouveaux éléments doivent être rendus visibles.

Le RecyclerView demande alors à l'adaptateur de lui instancier (*inflate*) les vues pour afficher les éléments.

Le nom « RecyclerView » vient de l'astuce : les vues qui deviennent invisibles à cause du défilement vertical sont recyclées et renvoyées de l'autre côté mais en changeant seulement le contenu à afficher.

Recyclage des vues

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



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

ViewHolders

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

Un *ViewHolder* est instancié pour chaque élément visible de la liste d'items, ex: une planète $\longrightarrow$ un *ViewHolder*.

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

Exemple de ViewHolder

D’abord, il faut un layout d’item, res/layout/planete.xml :

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout ...>
    <TextView android:id="@+id/nom" .../>
    <TextView android:id="@+id/distance" .../>
</RelativeLayout>
```

D’autres informations peuvent facilement être rajoutées : images...

Ce même layout sera instancié pour chaque planète visible dans le RecyclerView.

Les TextView seront affectés selon la planète associée.

On va utiliser son *ViewBinding*, c’est à dire la classe PlaneteBinding, pour accéder facilement aux TextView.

Exemple de ViewHolder

Voici la classe PlaneteViewHolder :

```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
{
    private final PlaneteBinding ui;

    public PlaneteViewHolder(PlaneteBinding ui) {
        super(ui.getRoot());
        this.ui = ui;
    }
    public void setPlanete(Planete planete) {
        ui.nom.setText(planete.getNom());
        ui.distance.setText(
                Integer.toString(planete.getDistance()));
    }
}
```

Exemple de ViewHolder

La classe PlaneteViewHolder mémorise un `PlaneteBinding`, c'est à dire l'ensemble des vues représentant une planète à l'écran, provenant de `res/layout/planete.xml`. Ce *ViewBinding* sera passé en paramètre par l'adaptateur[1] et mémorisé dans le *ViewHolder*.

La méthode `setPlanete` met à jour ces vues à partir de la donnée passée en paramètre. Cette méthode est appelée par l'adaptateur lors du recyclage.

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

[1]La création du *ViewBinding* est faite en amont, par l'adaptateur. On ne peut pas faire autrement car le constructeur de la superclasse *ViewHolder* demande une interface déjà créée.

Rôle d'un adaptateur

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

L'adaptateur est une classe qui :

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

On retrouve donc ces méthodes dans sa définition.

Définition d'un adaptateur

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

Par exemple, PlaneteAdapter :

```
public class PlaneteAdapter
        extends    RecyclerView.Adapter<PlaneteViewHolder>
{
    ... constructeur ...
    ... surcharge des méthodes nécessaires...
}
```

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

Constructeur d’un adaptateur

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

```
public class PlaneteAdapter
        extends    RecyclerView.Adapter<PlaneteViewHolder>
{
    private final List<Planete> liste;

    PlaneteAdapter(List<Planete>  liste)  {
        this.liste = liste;
    }
...
```

La liste est stockée dans l’adaptateur.
NB: c’est un partage de référence, il n’y a qu’une seule allocation en mémoire.

Méthodes à ajouter

Ensuite, pour communiquer avec le RecyclerView, il faut surcharger (redéfinir) trois méthodes. Pour commencer, celle qui retourne le nombre d'éléments :

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

Méthodes à ajouter

Ensuite, surcharger la méthode qui crée les ViewHolder :

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

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

inflate = transformer un fichier XML en vues Java.

Méthodes à ajouter

Enfin, surcharger la méthode qui recycle les ViewHolder :

```
@Override public void
    onBindViewHolder(PlaneteViewHolder holder, int position)
{
    Planete planete = liste.get(position);
    holder.setPlanete(planete);
}
```

Cette méthode est appelée pour remplir un ViewHolder avec l'un des éléments de la liste, celui qui est désigné par `position` (numéro dans la liste à l'écran). C'est très facile avec le *setter*.

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

Configuration de l’affichage

Optimisation du défilement

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

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

```
@Override

protected void onCreate(Bundle savedInstanceState)
{
   ...

   // dimensions constantes
   ui.recycler.setHasFixedSize(true);
```

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

LayoutManager

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

Cela se fait avec un *LayoutManager* :

```
...
// layout manager
RecyclerView.LayoutManager lm = new LinearLayoutManager(this);
ui.recycler.setLayoutManager(lm);
```

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

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

LayoutManager dans le layout.xml

Il est possible de spécifier le *LayoutManager* directement dans le fichier XML du layout :

```
<androidx.recyclerview.widget.RecyclerView
      xmlns:android="http://schemas.android.com/apk/res/android"
      android:id="@+id/recycler"
      android:layout_width="match_parent"
      android:layout_height="match_parent"
      xmlns:app="http://schemas.android.com/apk/res-auto"
      app:layoutManager=
            "androidx.recyclerview.widget.LinearLayoutManager"  />
```

Mais on ne peut pas le configurer aussi bien que par programmation.

LayoutManager

Un LinearLayoutManager organise les éléments en liste, en leur donnant tous la même taille. On peut le configurer pour afficher la liste horizontalement, mais il faut alors prévoir le layout des items en conséquence :

```
// layout manager liste
LinearLayoutManager lm =
  new LinearLayoutManager(this,
          RecyclerView.HORIZONTAL,      // direction
          false);                       // sens
ui.recycler.setLayoutManager(lm);
```

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

Disposition en tableau

Au lieu d'un LinearLayoutManager, on peut créer un GridLayoutManager qui arrange en tableau d'un certain nombre de colonnes indiqué en paramètre :

```
// layout manager tableau
GridLayoutManager lm = new GridLayoutManager(this, 2);
ui.recycler.setLayoutManager(lm);
```

On peut aussi choisir l'axe de défilement horizontal :

```
GridLayoutManager lm =
   new GridLayoutManager(this,
           2,                              // nb lignes ou colonnes
           RecyclerView.HORIZONTAL,        // direction
           false);                         // sens
```

Disposition en blocs empilés

Encore une autre disposition, elle empile des *ViewHolders* qui peuvent avoir des hauteurs (ou largeurs, selon la direction d'empilement) différentes :

```
// layout manager grille d'empilement
StaggeredGridLayoutManager lm =
   new StaggeredGridLayoutManager(
            2,                        // colonnes
            RecyclerView.VERTICAL);   // empilement
ui.recycler.setLayoutManager(lm);
```

Séparateur entre items

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

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

Actions sur la liste

Présentation

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

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

- modifications extérieures au RecyclerView, càd le programme Java modifie les données directement dans le `ArrayList`,
- modifications effectuées par le RecyclerView suite aux gestes de l'utilisateur sur les éléments (clics, glissés...).

Modification des données

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

Selon la modification, il faut appeler :

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

Défilement vers un élément

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

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

On peut configurer cette dernière pour avoir un meilleur comportement.

Clic sur un élément

Gros problème : dans Android, rien n'est prévu pour les clics sur les éléments. On doit construire soi-même une architecture d'écouteurs. Voici d'abord la situation, pour comprendre la solution.

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

Clic sur un élément

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

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

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

Notre écouteur de clics

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

```
public class PlaneteAdapter
              extends   RecyclerView.Adapter<PlaneteViewHolder>
{
  public interface OnItemClickListener {
    void onItemClick(int position);
  }

  private OnItemClickListener listener;

  public void setOnItemClickListener(OnItemClickListener l)
    this.listener = l;
  }
  ...
```

Notre écouteur de clics

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

```
public void onBindViewHolder(PlaneteViewHolder holder, ...)
{
  ...
  holder.setOnItemClickListener(this.listener);
}
```

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

Notre écouteur de clics

Les *ViewHolders* doivent mémoriser cet écouteur :

```
public class PlaneteViewHolder extends RecyclerView.ViewHolder
        implements View.OnClickListener
{
  private final PlaneteBinding ui;
  private PlaneteAdapter.OnItemClickListener listener;

  public void setOnItemClickListener(OnItemClickListener l) {
    this.listener = l;
  }
  ...
```

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

Notre écouteur de clics

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

```
public PlaneteViewHolder(@NonNull PlaneteBinding ui) {
  super(ui.getRoot());
  this.ui = ui;
  itemView.setOnClickListener(this);
}

@Override public void onClick(View v) {
  if (listener != null)
      listener.onItemClick(getAdapterPosition());
}
```

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

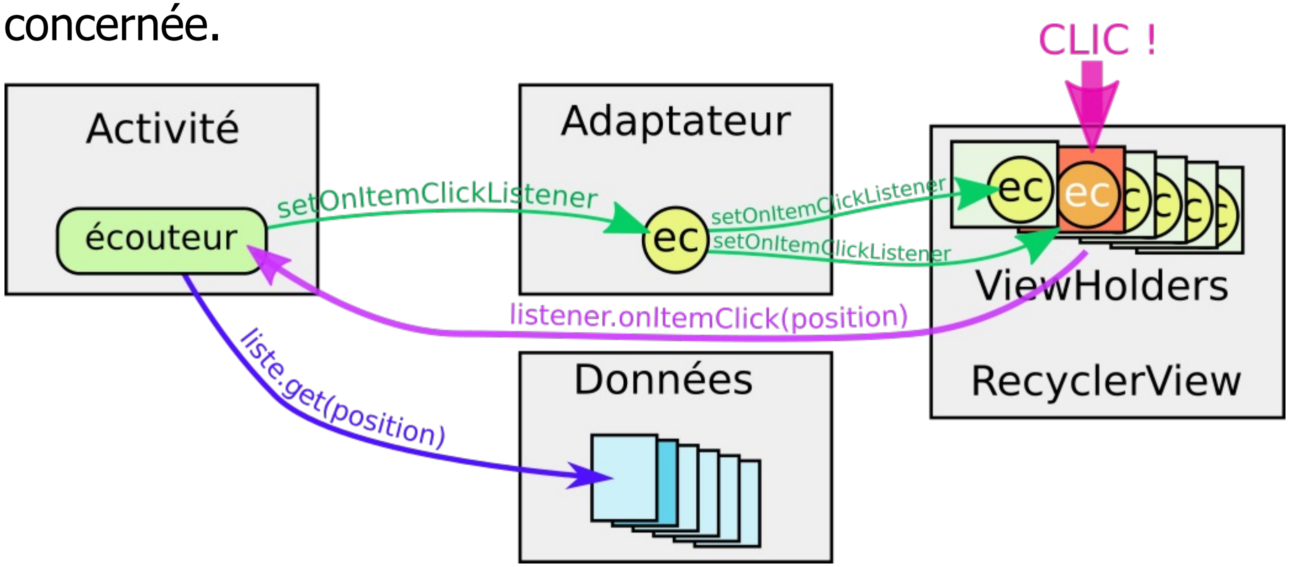
Notre écouteur de clics

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

```
public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        ...
        // adaptateur
        adapter = new PlaneteAdapter(liste);

        // écouteur pour les clics sur les éléments de la liste
        adapter.setOnItemClickListener(this::onItemClick);
    }
    private void onItemClick(int position) {
        Planete planete = liste.get(position);
        ... utiliser planete ...
    }
```

Schéma récapitulatif

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



Les Permissions

Introduction aux permissions

Qu'est-ce qu'une permission ?

- Une **permission** détermine ce qu'une application a le droit d'effectuer sur un périphérique.

Objectifs des permissions :

- Restreindre les actions d'une application.
- Protéger les données présentes sur l'appareil.

Par défaut :

- Les applications **ne peuvent pas** :
  - Accéder à d'autres applications.
  - Modifier le système.
  - Lire les données utilisateur.

Isolation des applications - Sandbox

Qu'est-ce qu'une sandbox ?

Chaque application fonctionne dans son propre espace sécurisé appelé **sandbox**.

Propriétés de la sandbox :

- Isolation complète des applications.
- Protection des **données utilisateur** et du **système**.

Validation des permissions :

- **Permissions normales** : Automatiquement acceptées.
- **Permissions dangereuses** : Validation obligatoire par l'utilisateur.

Les types de permissions

Permissions normales :

- **Faible risque** pour l'utilisateur.
- Exemples :
  - Allumer le **flash**.
  - Vérifier l'état de la **connexion Wi-Fi**.
- **Attribution automatique** sans intervention utilisateur.

Permissions dangereuses :

- Risque de compromettre :
  - Les **données personnelles**.
  - D'autres applications.
- Exemples :
  - Lire les **contacts**.
  - Envoyer des **SMS**.
- **Validation requise** par l'utilisateur.

Définir des permissions

Où déclarer les permissions ?
Les permissions sont définies dans le fichier **AndroidManifest.xml**.
Syntaxe :
•Utilisation de la balise <uses-permission>.
Exemple :

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
          package="fr.ulco.renaud.myapp">
    <!-- Permission normale -->
    <uses-permission android:name="android.permission.ACCESS_WIFI_STATE" />
    <!-- Permission dangereuse -->
    <uses-permission android:name="android.permission.RECEIVE_SMS" />
</manifest>
```

Permissions normales - Détails

Caractéristiques :

- Sans risque pour l'utilisateur.
- Elles sont automatiquement attribuées.

Exemples courants :

1. **ACCESS_WIFI_STATE** : Vérifier l'état du réseau Wi-Fi.
2. **FLASHLIGHT** : Allumer le flash.
3. **VIBRATE** : Utiliser la vibration du téléphone.

Permissions dangereuses - Détails

Caractéristiques :

- Risque élevé de compromettre les données.
- Validation obligatoire par l'utilisateur.

Exemples courants :

1. **READ_CONTACTS** : Lire les contacts de l'utilisateur.
2. **RECEIVE_SMS** : Lire les SMS reçus.
3. **ACCESS_FINE_LOCATION** : Accéder à la position GPS précise.

Scénario pratique

Exemple : Application de messagerie
Les permissions nécessaires incluent :

1.Permission normale :
1. ACCESS_NETWORK_STATE : Vérifier l'état réseau.

2.Permissions dangereuses :
1. READ_CONTACTS : Lire les contacts.
2. SEND_SMS : Envoyer des SMS.

Gestion responsable des permissions

Bonnes pratiques :

1. **Ne demander que les permissions nécessaires**.
2. **Informer l'utilisateur** sur les raisons de chaque demande.
3. **Gérer les refus de permissions** correctement dans l'application.

Conséquences d'une mauvaise gestion :

- Atteinte à la **confidentialité**.
- Mauvaise **expérience utilisateur**.
- Exposition à des **risques de sécurité**.

Historique des permissions (Android 5.1 vs 6.0+)

Jusqu'à Android 5.1 :

- Les permissions sont **validées à l'installation**.
- Elles sont **permanentes** une fois accordées.

À partir d'Android 6.0 (API 23) :

- **Permissions d'installation** : attribuées automatiquement.
- **Permissions d'exécution** :
  - Validées au **moment où elles sont requises** (par exemple, accès à la caméra).
  - L'utilisateur peut **retirer des permissions** à tout moment.

Quelques permissions courantes

Permissions normales :

- **ACCESS_NETWORK_STATE** : Vérifie l'état réseau.
- **ACCESS_WIFI_STATE** : Accède à l'état du Wi-Fi.
- **SET_ALARM** et **SET_WALLPAPER** : Configurer des alarmes et le fond d'écran.
- **VIBRATE** : Utiliser la vibration du téléphone.

Permissions dangereuses :

- **ACCESS_FINE_LOCATION** : Accéder à la position GPS précise.
- **CAMERA** : Utiliser la caméra du périphérique.
- **READ_EXTERNAL_STORAGE** : Lire les fichiers externes.

Vérification de la présence d'une permission

Méthode pour tester une permission :

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
    != PackageManager.PERMISSION_GRANTED) {
    // Gérer le cas où la permission est refusée
}
```

Fonction utilisée :

- ContextCompat.checkSelfPermission(Context, String)

Valeurs retournées :

- PackageManager.PERMISSION_GRANTED : Permission accordée.
- PackageManager.PERMISSION_DENIED : Permission refusée.

Bonnes pratiques pour gérer les permissions

1. Minimiser les demandes de permissions :

- Ne demander que les permissions **nécessaires** au bon fonctionnement.

2. Fournir une explication :

- Informer l'utilisateur pourquoi une permission est requise.
- Utiliser **shouldShowRequestPermissionRationale** pour expliquer.

3. Gérer le refus des permissions :

- Proposer une **alternative** ou informer des conséquences d'un refus.

Exemple de permission en contexte réel

Scénario : Une application de cartographie

•**Permissions requises :**

- **ACCESS_FINE_LOCATION** : Obtenir la position précise.
- **INTERNET** : Charger les cartes en ligne.
- **READ_EXTERNAL_STORAGE** : Accéder aux cartes hors ligne.

Interface utilisateur :

•Pop-up pour demander la permission.

•Paramètres pour retirer les permissions si nécessaire.

Comparaison des versions Android

Version Android	Gestion des permissions
Avant 6.0 (API 23)	Permissions accordées à l'installation.
Depuis 6.0 (API 23)	Permissions demandées à l'exécution.
Fonctionnalité clé	L'utilisateur peut retirer les permissions à tout moment.

Demande d'une permission

Utilisation de la méthode :
La méthode **ActivityCompat.requestPermissions()** permet de demander une ou plusieurs permissions à l'utilisateur.
Structure de la méthode :

```
public static void requestPermissions(
    Activity activity,
    String[] permissions,
    int requestCode
);
```

Paramètre	Description
Activity	Activité demandeuse (contexte `this`).
String[]	Liste des permissions (Manifest.permission).
int requestCode	Identifiant unique de la demande.

Demande d'une permission

Exemple 1 - Demande multiple de permissions

```
public final int MY_PERMISSIONS_REQUEST = 1;

String[] liste = {
    Manifest.permission.CAMERA,
    Manifest.permission.READ_EXTERNAL_STORAGE
};

ActivityCompat.requestPermissions(this, liste, MY_PERMISSIONS_REQUEST);
```

Explications :
MY_PERMISSIONS_REQUEST : Identifiant de la requête.
String[] liste : Liste des permissions à demander.
La méthode demande à l'utilisateur d'autoriser **CAMERA** et **READ_EXTERNAL_STORAGE**.

Demande d'une permission

Exemple 2 - Vérification et demande d'une permission unique

```java
public final int MY_PERMISSIONS_REQUEST_LOCATION = 2;

if (ContextCompat.checkSelfPermission(this, Manifest.permission.ACCESS_FINE_LOCATION
        != PackageManager.PERMISSION_GRANTED) {
    ActivityCompat.requestPermissions(
        this,
        new String[]{Manifest.permission.ACCESS_FINE_LOCATION},
        MY_PERMISSIONS_REQUEST_LOCATION);
} else {
    // Permission déjà accordée
}
```

Explications :
checkSelfPermission : Vérifie si la permission est accordée.
Si refusée → Appel à `requestPermissions` pour demander **ACCESS_FINE_LOCATION**.

Gestion de la réponse

Surcharge de la fonction callback
Pour gérer la réponse de l'utilisateur, on surcharge **onRequestPermissionsResult** :

```
public void onRequestPermissionsResult(
    int requestCode,
    String permissions[],
    int[] grantResults
) {
    // Logique de réponse
}
```

Paramètre	Description
requestCode	Identifiant de la demande.
permissions[]	Liste des permissions demandées.
grantResults[]	Résultat pour chaque permission (GRANTED/DENIED).

Gestion de la réponse

Exemple de traitement des résultats :

```
@Override
public void onRequestPermissionsResult(int requestCode, String permissions[], int[] grantResults) {
   switch (requestCode) {
      case MY_PERMISSIONS_REQUEST_LOCATION:
         if (grantResults.length > 0 && grantResults[0] == PackageManager.PERMISSION_GRANTED) {
            // Permission accordée → Utilisation du GPS
         } else {
            // Permission refusée → Gérer le refus
         }
         break;
   }
}
```

Explications :
Vérifie si la réponse est **PERMISSION_GRANTED**.
Gère le cas où la permission est refusée (**else**).

Classe de contrat pour simplifier

Utilisation d'ActivityResultContracts.RequestPermission

```
private ActivityResultLauncher<String> simplePermissionLauncher =
    registerForActivityResult(new ActivityResultContracts.RequestPermission(),
        new ActivityResultCallback<Boolean>() {
            @Override
            public void onActivityResult(Boolean res) {
                if (res) {
                    // Permission accordée
                } else {
                    // Permission refusée
                }
            }
        });
```

Avantages :
Plus besoin d'utiliser un **requestCode**.
Résultat directement dans **onActivityResult**.

Lancer une demande avec la classe de contrat

Exemple : Lancement de la demande

```
if (ContextCompat.checkSelfPermission(this, Manifest.permission.CAMERA)
        != PackageManager.PERMISSION_GRANTED) {
    simplePermissionLauncher.launch(Manifest.permission.CAMERA);
} else {
    // Permission déjà accordée
}
```

Explication :
checkSelfPermission : Vérifie si la permission est accordée.
launch : Lance la demande pour **CAMERA**.

Demande de permissions multiples

Classe ActivityResultContracts.RequestMultiplePermissions

```
private ActivityResultLauncher<String[]> multiplePermissionLauncher =
  registerForActivityResult(new ActivityResultContracts.RequestMultiplePermissions(),
    new ActivityResultCallback<Map<String, Boolean>>() {
      @Override
      public void onActivityResult(Map<String, Boolean> res) {
        if (res.get(Manifest.permission.CAMERA)) {
          // Permission CAMERA accordée
        }
        if (res.get(Manifest.permission.ACCESS_FINE_LOCATION)) {
          // Permission LOCATION accordée
        }
      }
    });
```

Paramètres :
String[] : Liste des permissions demandées.
Map<String, Boolean> : Résultats par permission.

Lancement d'une demande multiple

Exemple complet :

```
String[] listePermission = {
    Manifest.permission.ACCESS_FINE_LOCATION,
    Manifest.permission.CAMERA
};
multiplePermissionLauncher.launch(listePermission);
```

Explications :
listePermission : Liste des permissions à demander.
multiplePermissionLauncher.launch : Lance la demande.