1. Introduction

On continue l'étude d'Android Views qui est la manière traditionnelle de créer une application, par opposition à Jetpack Compose qu'on verra dans les prochains TP.

Une application Android est généralement composée d'« activités ». Une activité Android est une interface utilisateur qui remplit tout l'écran, associée à une classe Kotlin qui gère les actions de l'utilisateur. La structure de l'interface est définie par un fichier appelé *layout*. C'est un fichier XML qui déclare chaque composant de l'interface, textes, boutons, etc, voir TP2. Donc, pour créer une activité, on définit un *layout* et une classe Kotlin.

Dans un *layout*, il y a des composants : textes, boutons, images, etc. Lorsque l'activité est affichée, chaque composant existe sous forme d'un objet Kotlin, ce qui permet de le manipuler à l'aide de ses *setters* et *getters*, et d'autres méthodes.

On va partir d'un projet minimal, type « Empty Views Activity », nommez-le `TP3`.

2. Ressources

Ce sont les fichiers XML qui sont dans *projet*`/src/main/res`. Il y en a de plusieurs sortes, dont :

- `res/values` : le fichier `strings.xml` contient les textes de votre application : labels, messages, etc. Ces textes sont identifiés par l'attribut `name` et référencés dans les layouts par `@string/nom`, et par `R.string.nom` dans les sources Kotlin. Voir le TP2 pour un exemple de traduction des textes.
- `res/drawable` et `res/mipmap`, ce sont des images, icônes et autres dans plusieurs résolutions. On ne s'y intéressera pas cette semaine.
- `res/layout` : ce sont les définitions d'écrans des activités, par exemple `activity_main.xml`. Les balises XML et leurs attributs définissent les éléments affichés : boutons, zones de saisie, etc. Dans le TP3, on va se concentrer sur ces fichiers.

Remarque : les fichiers XML peuvent être affichés soit sous forme d'un formulaire ou de manière graphique, soit sous forme XML brute. Voyez l'onglet en haut à droite de l'éditeur, ex: `Code`, `Split` ou `Design`. Le mode Code offre un contrôle total. Il est préférable au mode graphique quand on sait ce qu'on fait. NB: dans l'interface actuelle, il n'y a que les icônes.

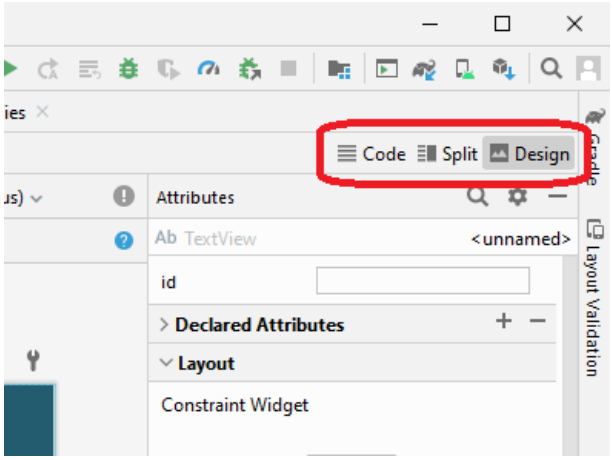

Figure 1: Mode Code, Split ou Design

2.1. Ressources, identifiants et références

Avant de commencer, quelques notions très importantes. La plupart des éléments des ressources sont identifiés. Par exemple, tous les textes présents dans `res/values/strings.xml` :

```
<string name="identifiant">texte</string>
...
```

L'attribut `name` définit **l'identifiant de la ressource chaîne**.

Dans le TP2, vous avez vu que, dans le fichier `res/layout/activity_main.xml`, quand on avait extrait les chaînes des layouts, le texte avait été remplacé par `@string/identifiant` :

```
<TextView android:text="@string/identifiant"
    ... />
```

La balise `TextView` représente un texte non éditable qui est défini par l'attribut `android:text`. La syntaxe `@string/identifiant` est une **référence de ressource chaîne**. Il y a aussi des références de couleurs, par exemple `@color/teal_700`, de style, etc.

Ne confondez pas *identifiant* (= définition) et *référence d'identifiant* (= utilisation).

Dans certains layouts, il faut identifier les vues. Ça consiste à rajouter un attribut spécial `android:id="@+id/nom"` :

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

La valeur d'attribut `@+id/titre` définit l'identifiant « titre » attribué au `TextView`.

Ensuite, dans une autre vue, on peut référencer cet identifiant avec une syntaxe subtilement différente, sans le `+` :

```
<Button
    android:layout_below="@id/titre"
    ... />
```

Se souvenir que :

- `@+id/nom` définit un identifiant unique,
- `@id/nom` est une référence à cet identifiant.

2.2. Ressources et classe `R`

Une interface utilisateur Android est définie par un fichier XML placé dans `res/layout`. Le nom de ce fichier est une sorte d'identifiant pour l'interface qu'il définit. Regardez dans `MainActivity.kt`, il y a une ligne `setContentView(R.layout.activity_main)`. C'est cette instruction qui indique à l'activité quelle est l'interface qu'elle doit afficher, celle de `activity_main.xml`.

La syntaxe `R.layout.nomdulayout` est très particulière. `R` est une classe générée automatiquement à partir de tout ce qui est trouvé dans les ressources. Voici comment elle serait programmée en Java :

```
public final class R {
    public static final class string {
        public static final int app_name=0x7f080000;
        public static final int titre=0x7f080001;
    }
    public static final class layout {
        public static final int activity_main=0x7f030000;
    }
    public static final class menu {
        public static final int main_menu=0x7f050000;
        public static final int context_menu=0x7f050001;
    }
    ...
```

Cette classe `R` associe des entiers à chaque ressource identifiée : chaînes, interfaces, éléments d'interface, icônes, etc, chacun dans une catégorie. Ainsi tous les layouts de `res/layout/*` sont dans `R.layout.*`, tous les identifiants de chaînes de `res/values/strings.xml` sont dans `R.string.*`, etc.

La classe `R` est générée de manière invisible par AndroidStudio. On ne peut pas voir son code source.

3. Layout et vues

On va maintenant approfondir l'étude des layouts.

Un layout contient, sous forme de balises XML la description de l'interface, c'est à dire les « vues » ou composants d'interface à mettre dedans et comment les placer les unes par rapport aux autres.

3.1. Arbre des vues

Il y a deux types de vues.

- Les « vues simples », comme `TextView`, `Button`, `CheckBox`, etc. Il y en a un grand nombre. Ces vues sont simples parce qu'elles ne contiennent aucune autre vue à l'intérieur.
- Les « vues de rangement », comme `LinearLayout`, `ConstraintLayout`, etc. Au contraire des vues simples, les vues de rangement contiennent d'autres vues à l'intérieur, et leur but est par exemple de les aligner automatiquement, ou selon des critères dynamiques.

Toutes ces vues d'une interface forment un arbre (*Component Tree*) dont les branches sont les vues de rangement, et les feuilles sont les vues simples.

Voici l'arbre de l'interface du TP2 :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
    ...
    android:id="@+id/main"
    android:orientation="vertical">
```

```
    <TextView android:text="Racine carrée" .../>

    <LinearLayout
        ...
        android:orientation="horizontal">

        <EditText android:id="@+id/nombre" .../>

        <Button android:text="calculer" .../>

    </LinearLayout>

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

</LinearLayout>
```

La racine de l'arbre est un `LinearLayout` vertical, il contient trois feuilles, un `TextView`, un autre `LinearLayout` et un `TextView`.

La figure 2 montre les trois vues, un `LinearLayout` qui contient un `TextView` et un `Button`.

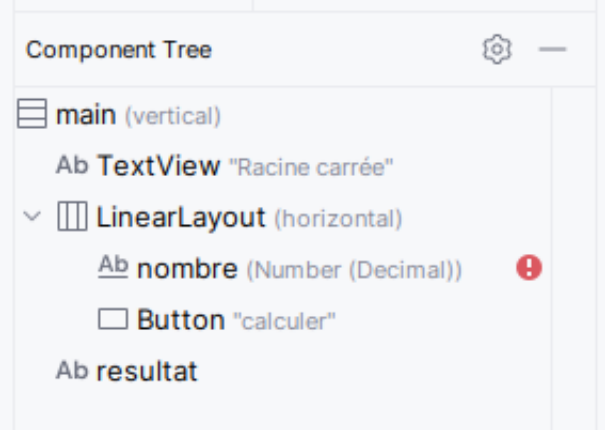

Figure 2: Arbre de vues

3.2. Paramétrage des vues

Les vues d'une interface ont de très nombreuses propriétés : titre, couleur, taille, etc. Ces propriétés peuvent être spécifiées par programmation, à l'aide de *setters*, mais elles sont aussi définies dans le fichier XML sous forme d'attributs.

```
    <TextView
        android:id="@+id/textview"
        ...
        android:textSize="28dp"
        android:textColor="@color/teal_700"
        android:text="Bonjour"
```

```
        android:padding="10dp"
        android:gravity="center"/>
```

Il faut du temps pour apprendre toutes les propriétés utiles pour construire des interfaces agréables.

3.3. Attributs obligatoires

Il y a seulement deux attributs absolument obligatoires pour chacune des vues de l'interface, `android:layout_width` et `android:layout_height`. Ces deux propriétés définissent la taille souhaitée à l'écran. Le layout est invalide s'il manque ces informations sur l'une des vues, y compris la vue racine.

Ces deux informations, largeur et hauteur, peuvent avoir ces valeurs :

- `match_parent` : signifie que la vue souhaite adopter la taille (largeur ou hauteur) de l'élément de rangement parent,
- `wrap_content` : signifie que la vue veut être la plus ajustée possible,
- `NNNdp` : la vue veut une taille de NNN pixels. Les `dp` sont des pixels dont la taille est indépendante de l'écran. 1 dp = 1 pixel sur un écran 160 dpi. Si l'écran est en 320 dpi, alors 1 dp = 2 pixels. C'est pour éviter d'avoir des interfaces qui changent de taille d'un smartphone à l'autre, selon la finesse des pixels de l'écran.
- `0dp` : signifie que la place attribuée à la vue par son parent dépendra des autres vues et d'un autre attribut appelé `android:layout_weight`. Voir les exercices plus loin.

3.4. Composants à connaître

Android contient plusieurs dizaines de composants d'interface :

- des vues : `TextView`, `Button` et `EditText` ...
- des groupes : `LinearLayout`, `TableLayout`... lire declaring-layout.html.

Il faut savoir que les composants évoluent et peuvent être déclarés obsolètes, par exemple `DigitalClock`, `Gallery`...

On vous invite à explorer ces documentations, avec curiosité.

4. Tutoriel sur les groupements de vues

On va s'intéresser aux vues qui permettent de positionner d'autres vues. Elles dérivent de la classe `ViewGroup`. Pour simplifier, elles ont des vues enfants et décident de leurs tailles et positions, en fonction de propriétés présentes sur les enfants. Voici les deux à connaître :

- `LinearLayout` pour former des lignes ou des colonnes, voir doc
- `ConstraintLayout` pour des dispositions dynamiques complexes.

4.1. Mise en page avec des `LinearLayout`

C'est le composant de rangement le plus simple et le plus facile à employer. Il dispose ses vues enfants soit en ligne, soit en colonne. Il n'est conseillé que pour des interfaces simples.

Téléchargez ce document XML et appelez-le `res/layout/linear.xml` :

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

    <Button
        android:text="OK"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:text="Annuler"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>
    <Button
        android:text="Tout supprimer"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"/>

</LinearLayout>
```

Examinez la mise en page en comparant avec les propriétés des vues, puis faites les changements suivants :

- Mettez `match_parent` pour `android:layout_width` de l'un des boutons, puis annulez le changement.
- Mettez `match_parent` pour `android:layout_height` du deuxième bouton, regardez l'effet, puis annulez. Vous avez vu que le 3e bouton n'est plus visible.

Retenir : `match_parent` signifie « prendre toute la place restante » et `wrap_content` signifie « prendre la place minimale ».

- Mettez `200dp` puis `300dp` pour `android:layout_height` du deuxième bouton, regardez son effet, puis annulez.
- Changez `android:orientation` en `horizontal` pour le `LinearLayout`, laissez ce changement. Passez en mode paysage si les vues sont trop encombrantes.

On voit que les vues ont une largeur dépendant de leur contenu. On voudrait qu'elles aient la même largeur.

- Ajoutez `android:layout_weight="1"` pour chacune. C'est pas très visible, mais la largeur n'est pas exactement la même à cause des titres.
- Changez `android:layout_width="0dp"` pour les trois vues. C'est ce qui donne le meilleur résultat : largeur nulle combinée avec un poids non nul.
- Changez `android:layout_width="wrap_content"` pour le `LinearLayout` puis annulez ce changement.

Maintenant, à vous. Vous devez obtenir ceci, figure 3 :

Figure 3: Saisie d'un mot de passe

1. Copiez `linear.xml` et appelez-le `password.xml`
2. Supprimez les 2e et 3e boutons, ne laissez que le `OK`.
3. Ajoutez un `EditText` devant le bouton `OK`.
4. Remplacez la propriété `android:text` par `android:hint` et mettez la chaîne `"mot de passe"`.
5. Puis modifiez les largeurs pour obtenir le résultat demandé. C'est à dire que l'`EditText` prend presque toute la place en largeur, tandis que le bouton `OK` est le plus petit possible. Pour cela, il faut mettre un poids nul au bouton `OK` mais aussi lui donner une taille minimale.

4.2. Mise en page avec un `ConstraintLayout`

Ce type de `ViewGroup` permet de mettre à jour la disposition de manière dynamique et polyvalente. Le principe est de dire pour chaque vue, où on doit la mettre sur l'écran : en dessous de telle autre, à droite de telle autre... On définit donc un ensemble de contraintes de placement qu'Android essaye de satisfaire. Paradoxalement, un `ConstraintLayout` est conseillé pour des interfaces complexes, parce que les autres layouts sont beaucoup plus lents pour faire la mise en page.

Plus précisément, dans un `ConstraintLayout`, on doit spécifier chaque bord d'une vue. Les bords sont le haut (`Top`), le bas (`Bottom`), le côté gauche (`Left`) et le côté droit (`Right`). Si on spécifie, par exemple, le côté gauche et la largeur, alors le côté droit s'en déduit. Donc par exemple, on peut indiquer la hauteur et la largeur avec `wrap_content` et seulement indiquer sur quoi sont alignés les bords haut et gauche d'une vue.

Les alignements des côtés sont spécifiés avec les attributs des vues. Ce qui fait peur au début, c'est qu'il y a en général 4 à 6 attributs pour chaque vue. Au minimum, il y a les alignements gauche et haut avec la largeur et hauteur. Au maximum, il y a les alignements de tous les côtés et en plus les largeurs et hauteurs valant `0dp`.

Heureusement, ça devient plus simple une fois qu'on a compris le principe.

Pour commencer, on rappelle que `layout_width` et `layout_height` sont des attributs obligatoires. On doit les spécifier avec les valeurs suivantes : `wrap_content` pour la taille minimale acceptable, `match_parent` pour la taille maximale possible ou `0dp` pour laisser les contraintes décider.

Ensuite, les attributs pour spécifier les contraintes d'alignement ont des noms compliqués. La forme générale des contraintes est `app:layout_constraintXXX_toYYYOf` avec *XXX* et *YYY* valant `Top`, `Bottom`, `Left`, `Right`. Cela donne des attributs comme `app:layout_constraintTop_toTopOf`, `app:layout_constraintRight_toLeftOf`, etc.

Ils signifient que le coté *XXX* de cette vue doit s'aligner avec le côté *YYY* d'une autre vue, celle qui est dans la valeur de l'attribut.

Par exemple, `app:layout_constraintLeft_toRightOf` veut dire d'aligner le côté gauche de cette vue sur le côté droit d'une autre vue. L'autre vue est indiquée par son identifiant. Par exemple,

on peut écrire `app:layout_constraintTop_toBottomOf="@id/titre"` pour placer cette vue en dessous du titre. Quand l'autre vue est l'écran lui-même, alors on met une valeur spéciale : `"parent"`.

Voici maintenant un exemple à télécharger sous le nom `res/layout/constraint.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <Button
        android:id="@+id/button1"
        android:text="BTN1"
        android:backgroundTint="#ff683e"
        app:cornerRadius="8dp"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toTopOf="parent"
        android:layout_width="wrap_content"
        app:layout_constraintLeft_toLeftOf="parent"
        app:layout_constraintRight_toRightOf="parent"/>
    <Button
        android:id="@+id/button2"
        android:text="BTN2"
        android:backgroundTint="#19C7C1"
        app:cornerRadius="8dp"
        android:layout_height="wrap_content"
        app:layout_constraintTop_toBottomOf="@id/button1"
        android:layout_width="0dp"
        app:layout_constraintLeft_toLeftOf="@id/button1"
        app:layout_constraintRight_toRightOf="parent"/>

</androidx.constraintlayout.widget.ConstraintLayout>
```

Le premier bouton est positionné en haut. Le second bouton est mis sous le premier, et comme sa largeur est `0dp`, elle est définie par ses contraintes : du bord gauche du bouton 1 au bord droit du parent.

Testez les modifications suivantes (CTRL-Z après chacune) :

- Bouton 1 : `android:layout_width="match_parent"` au lieu de `"wrap_content"` Vous voyez que les deux boutons prennent instantanément toute la largeur. Le 2e bouton aligne son côté gauche sur celui du bouton 1.
- Bouton 2 : attribut `app:layout_constraintLeft_toLeftOf` changez en `app:layout_constraintLeft_toRightOf`
- Bouton 2 : `app:layout_constraintRight_toRightOf="@id/button1"` au lieu de `"parent"`
- Bouton 2 : `android:layout_width="wrap_content"` au lieu de `"0dp"`

À vous maintenant. Essayez de reproduire cette disposition avec un `ConstraintLayout` :

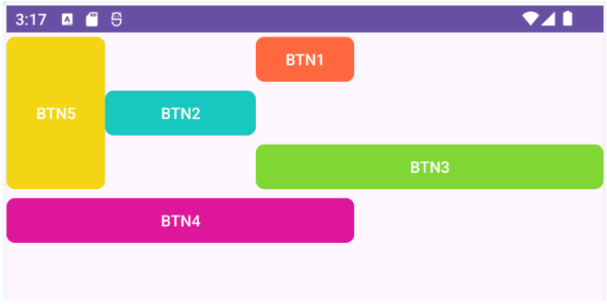

Figure 4: Résultat à obtenir avec `constraint.xml`

Le bouton 1 ne change pas. Le bouton 2 est placé en bas et à gauche du n°1 et son côté gauche est aligné avec le côté droit du bouton 5 (le bouton 2 s'étire horizontalement). Le bouton 3 est sous le n°2 et va horizontalement du bord droit du bouton 2 au bord droit de la fenêtre. Et ainsi de suite. Vous constatez que le bouton 5 est référencé par le bouton 2 mais il est créé après. Et le bouton 5 a besoin du bouton 3 qui dépend du 2. C'est le mécanisme de satisfaction des contraintes qui résoud ce genre de boucle.

Pour positionner une vue à une certaine distance d'une autre, on utilise les attributs `layout_marginXXX`. Deux autres contraintes existent : `layout_constraintZZZ_bias` avec *ZZZ* étant `Vertical` ou `Horizontal`. Elles permettent de positionner une vue de manière proportionnellement à l'espace disponible – elles déséquilibrent une contrainte en faveur de l'un ou l'autre côté. La valeur à mettre est un réel entre 0 et 1, par défaut, c'est *0.5*.

5. Activités

On s'intéresse maintenant à la partie programme d'une application Android. Les interfaces sont gérées par des *activités*. Une activité est une classe contenant des écouteurs pour rendre l'interface active.

👉 Remplacez le contenu de `activity_main.xml` par ceci :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".MainActivity">
    <TextView
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Hello World!"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:layout_marginTop="20dp"
        android:layout_marginBottom="20dp"
```

```
        android:gravity="center"/>
    <Button
        android:id="@+id/bouton1"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_gravity="center"
        android:text="Cliquez moi (btn 1)"/>
</LinearLayout>
```

On va également faire un changement dans le thème de l'application pour faire apparaître un titre dans une barre d'outils, mais qui sera vide dans ce TP.

👉 Ouvrez le fichier `res/values/themes/themes.xml`, le premier des deux et remplacez dans la ligne 3, `parent="Theme.Material3.DayNight.NoActionBar"` par seulement `parent="Theme.Material3.DayNight"`. Ça permet d'afficher la *action bar* en haut de chaque activité. C'est une zone optionnelle où on trouve des menus.

👉 Lancez cette application sur un AVD.

NB: s'il y a un problème de compilation, c'est parce que l'assistant du SDK crée un projet trop récent pour la version d'Android Studio installée. Uniquement dans ce cas, faites ceci :

1. Dépliez la catégorie `Gradle Scripts` dans le projet
2. Ouvrez `libs.versions.toml`. Ce fichier contient les numéros de version des dépendances. Deux d'entre elles ne sont pas bonnes, `coreKtx` et `activity`.
3. Modifiez ces deux lignes comme ceci :

```
coreKtx = "1.13.1"
activity = "1.9.0"
```

4. Cliquez sur `Sync Now` en haut.
5. Tentez de relancer l'application sur l'AVD.

5.1. Classe `MainActivity`

👉 Affichez la classe `MainActivity`

Une activité Android doit être une sous-classe de `Activity`. Ici, `MainActivity` est même une sous-classe de `AppCompatActivity` qui dérive de `Activity`.

Une activité doit obligatoirement surcharger la méthode `onCreate`. Cette méthode est appelée lorsque l'activité démarre. Son rôle est de mettre en place l'interface.

👉 Modifiez `onCreate` de cette manière (il y aura un *import* à ajouter pour `Log`) :

```
    private val TAG = "TP3"

    override fun onCreate(savedInstanceState: Bundle?) {
        // initialisation interne de l'activité
        super.onCreate(savedInstanceState)
        // mise en place du layout activity_main
```

```
        setContentView(R.layout.activity_main)
        // titre de l'activité dans l'ActionBar
        setTitle(localClassName)
        // message d'information
        Log.i(TAG, "dans ${localClassName}.onCreate")
    }
```

Que déclare le mot-clé `override` (`@Override` en Java) ? Il indique que la méthode concernée doit impérativement exister dans la superclasse, donc que c'est véritablement une surcharge. Pour voir, changez `onCreate` en `onCreation` et vous verrez que le mot-clé `override` sera en erreur. Ça permet d'éviter des erreurs, en croyant surcharger une méthode, à tort quand on se trompe dans le nom ou les paramètres.

Il y a une autre particularité de Kotlin. Le symbole `localClassName` est un raccourci pour `this.getLocalClassName()`. Kotlin transforme les *getters* en pseudo variables membres. Ainsi quand on écrit `tv.text`, en fait, c'est `tv.getText()`. Ça a été présenté à la fin du TP2.

5.2. Cycle de vie d'une activité

Il y a d'autres méthodes similaires à `onCreate` qu'on peut surcharger. Elles sont liées au *cycle de vie* des activités, c'est à dire à ce qui se passe lors de la création et la destruction d'une activité. Par exemple, chaque fois qu'une activité arrive au premier plan, le système Android appelle sa méthode `onCreate`.

☛ Ajoutez ces méthodes : 

```
    override fun onDestroy() {
        super.onDestroy()
        Log.i(TAG, "dans ${localClassName}.onDestroy")
    }

    override fun onPause() {
        super.onPause()
        Log.i(TAG, "dans ${localClassName}.onPause")
    }

    override fun onResume() {
        super.onResume()
        Log.i(TAG, "dans ${localClassName}.onResume")
    }
```

☛ Relancez l'application sur l'AVD, cherchez le message concernant `onCreate` dans le LogCat (filtrez l'affichage pour ne voir que les messages ayant le *tag* TP3). Vous voyez aussi que `onResume` a été appelée.

☛ Appuyez sur le bouton rond ◯ pour revenir au lanceur (bureau Android), puis le bouton carré pour revenir dans l'application. Cherchez les messages correspondant pour savoir quelles méthodes ont été appelées dans quel ordre et à quel moment.

☛ Appuyez sur le bouton *back* ◁, cherchez le message concernant `onDestroy`. Vous verrez que `onPause` a été appelée juste avant.

☛ Relancez l'application puis faites pivoter l'écran (portrait → paysage et inverse). Constatez que le système Android détruit carrément l'activité, puis la re-crée ! C'est à savoir : on ne peut pas maintenir des variables membres longtemps dans une activité. Elles sont réinitialisées à chaque mouvement du téléphone ou lancement d'une autre application. Ces informations doivent être stockées autrement.

5.3. Manifeste

Toute activité doit être déclarée dans le fichier `AndroidManifest.xml`, qui est dans la catégorie `manifests` du projet.

☛ Ouvrez ce fichier et étudiez les balises `<application>` et `<activity>`. Le nom de la classe est mis dans un attribut `android:name=".MainActivity"`. Le point devant le nom de la classe est une abréviation du *package*.

Remarquez la balise `<intent-filter>` et son contenu. Cette balise sert à désigner cette activité comme étant la principale, celle qu'il faut lancer quand on démarre l'application. Le problème ne se pose pas ici car il n'y a qu'une seule activité.

5.4. Lien entre layout et activité

Dans le TP2, on a vu deux méthodes pour lier le layout à l'activité.

- en utilisant les identifiants des vues :

```
override fun onCreate(savedInstanceState: Bundle?) {
    // initialisation interne de l'activité
    super.onCreate(savedInstanceState)
    // mise en place du layout activity_main
    setContentView(R.layout.activity_main)
    ...
}

fun quelconque() {
    val tvMessage = findViewById<TextView>(R.id.message)
    tvMessage.text = "un autre message"
}
```

- en utilisant des *View Bindings*, beaucoup plus simple, fiable et efficace :

```
lateinit var ui: ActivityMainBinding

override fun onCreate(savedInstanceState: Bundle?) {
    // initialisation interne de l'activité
    super.onCreate(savedInstanceState)
    // mise en place du layout par un view binding
```

```
    ui = ActivityMainBinding.inflate(layoutInflater)
    setContentView(ui.root)
    ...
}

fun quelconque() {
    ui.message.text = "un autre message"
}
```

☛ Relisez le TD2 et faites la mise en place de l'interface du TP3 par un *View Binding*.

5.5. Attacher un écouteur à une vue

On voudrait maintenant qu'un clic sur le bouton change le message affiché. Pour cela, il faut que le bouton déclenche l'appel d'une méthode dans l'activité : un *écouteur*. Cet écouteur modifiera le message affiché.

Il y a plusieurs manières de définir un écouteur.

5.5.1. Définition d'un écouteur par ajout d'un attribut dans le layout

La manière la plus simple consiste à ajouter un attribut `android:onClick` au bouton dans le layout.xml. Cet attribut donne le nom de la méthode à appeler quand il y aura un clic. Voir le TP2.

On ne refera pas cette manipulation ici.

Il faut savoir que cette façon de faire est dépréciée parce qu'elle oblige le système Android à chercher la méthode dans la classe compilée (on appelle ça l'*introspection*). D'autre part, la méthode doit être publique, ce qui peut poser problème.

5.5.2. Définition d'un écouteur privé anonyme

Une deuxième manière pour définir un écouteur consiste à créer une classe anonyme avec une syntaxe un peu particulière en Kotlin, une *lambda*. C'est une écriture pour définir une méthode sans lui donner de nom : `(paramètres) -> {` corps de la méthode `}`.

☛ Rajoutez ceci à la fin de la méthode `onCreate` :

```
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        // écouteur pour le bouton1, lambda
        ui.bouton1.setOnClickListener {
            compteur += 1
            ui.message.text = "compteur = ${compteur}"
        }
    }

    var compteur = 0
```

La syntaxe Kotlin `objet.setChose { ... }` est un raccourci pour `objet.setChose(it -> { ... })`. C'est à dire que ça construit une *lambda* à partir d'un bloc `{...}` et en lui passant un paramètre appelé `it`, un peu comme `this` mais pour désigner l'objet passé en paramètre de la lambda. Ici, `it` représente le bouton mais n'est pas utilisé.

Cette technique consiste donc à définir une *lambda* pour chaque composant actif de l'interface et toutes les placer dans la méthode `onCreate` de l'activité. Cette façon de faire peut convenir à de toutes petites actions lors des clics. Il y aura un problème de lisibilité et de modularité s'il y a de nombreux composants à gérer ainsi, et/ou si les actions sont complexes.

En plus, ça sera impossible de faire des tests unitaires sur les actions des boutons.

D'autre part, une *lambda* ne peut être employée que lorsqu'il n'y a qu'une seule méthode à définir. Or certains composants demandent un écouteur définissant deux méthodes, et on ne peut pas le faire comme précédemment. En fait, la *lambda* précédente représente la syntaxe suivante, plus générale, dite de la *classe privée anonyme* qu'on retrouve en Java aussi.

1. Dupliquez le bouton `bouton1` dans le layout et, au nouveau, donnez-lui l'identifiant `bouton2` et adaptez son texte.
2. Rajoutez ceci à la fin de la méthode `onCreate` :

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // écouteur pour le bouton2, classe privée anonyme
    ui.bouton2.setOnClickListener(object : View.OnClickListener {
        override fun onClick(v: View?) {
            compteur += 2           // ok, compteur est la variable membre
            //this.compteur += 2    // AIE : ici this n'est pas l'activité
            //this@MainActivity.compteur += 2    // ça, c'est correct
            ui.message.text = "compteur = ${compteur}"
        }
    })
}
```

Cette syntaxe Kotlin est assez complexe. La base est `objet1.setChose(object : Classe)` qui signifie de créer un nouvel objet de la classe (ou interface) indiquée et de le fournir au setter de l'objet1. Ensuite, la classe est suivie d'accolades qui permettent de surcharger des méthodes : `objet1.setChose(object : Classe { méthodes surchargées... })`. Ici, il n'y a que `onClick`.

Attention avec ce genre de code : `this` ne désigne pas l'activité, mais l'écouteur privé anonyme. Décommentez la ligne `this.compteur += 3` pour constater que ça ne se compile pas. Il y a une solution, c'est la 3e ligne, avec `this@MainActivity` pour désigner le bon `this`.

Cette technique, de l'écouteur privé anonyme, avec sa syntaxe très compliquée, est à réserver à des situations vraiment spécifiques où on ne peut pas faire autrement.

5.5.3. Définition d'une référence de méthode

La meilleure manière de faire est probablement avec les *références de méthodes*. Cela consiste à définir une méthode pour gérer chaque événement (clic ou autre), et à définir un écouteur qui

est une *référence* à cette méthode. Une référence de méthode est un objet qui ne contient qu'une seule méthode, celle indiquée dans la référence.

On écrit `this::methode` pour créer un tel objet, `methode` étant le nom de l'une des méthodes de la classe, mis sans aucun paramètre. On peut mettre cet objet dans une variable par `val meth = this::methode` puis l'exécuter par `meth(params...)`. On peut employer un tel objet partout où il y a une lambda demandée en paramètre.

1. Dupliquez `bouton2` dans le layout avec l'identifiant `bouton3` et adaptez son texte.
2. Rajoutez ceci à la fin de la méthode `onCreate` :

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    // écouteur pour le bouton3, référence de méthode
    ui.bouton3.setOnClickListener(this::onBouton3Click)
}
```

3. Définissez plus loin la méthode `onBouton3Click` comme ceci :

```
fun onBouton3Click(view: View?) {
    compteur *= 2
    ui.message.text = "compteur = ${compteur}"
}
```

La syntaxe `this::nom_de_methode` appelée « référence de méthode » fait créer une sorte d'objet caché, n'ayant que la méthode attendue par `setOnClickListener`. Cette méthode sera appelée lors d'un événement.

La seule exigence, c'est qu'elle ait un paramètre de type `View`, même s'il ne sert pas.

L'avantage de cette technique sur les autres, c'est que la méthode peut être privée ou pas. La méthode peut être placée n'importe où dans le source. Il n'y a aucune syntaxe bizarre à part le paramètre `View`. Également, la méthode a librement accès aux variables membres de `this`, ce qui l'est pas forcément le cas des écouteurs privés anonymes. La modularité de cette technique est maximale. Elle peut faire l'objet de tests unitaires. Il n'y a aucun coût à l'exécution (pas de recherche dans un layout...). C'est clairement la solution qui a le plus de qualités.

6. Activités multiples

On va maintenant découvrir les interactions entre différentes activités.

1. Création d'une nouvelle activité :

- Créez une nouvelle activité appelée `InfosActivity` en utilisant le menu contextuel du projet : cliquez droit sur `app`, item `New`, puis sous-item `Activity`, et enfin `Empty Views Activity` (3 niveaux de menu !). Elle aura un layout appelé `activity_infos.xml`.

  Attention si vous vous trompez de type d'activité et que vous partez sur du Jetpack Compose, alors votre projet entier sera ruiné à cause de dépendances impossible à enlever. Vous pourrez recommencer le TP de zéro.

- Refaites l'attribution du layout par *view bindings* (`ActivityInfosBinding`) :
  - Définissez une variable membre `lateinit var ui: ActivityInfosBinding`
  - Remplacez `setContentView(R.layout.activity_infos)` par :

```
// mise en place du layout par un view binding
ui = ActivityInfosBinding.inflate(layoutInflater)
setContentView(ui.root)
// titre de l'activité
setTitle(localClassName)
```

2. Vérifiez le manifeste :

- Cette nouvelle activité a bien été ajoutée dans `AndroidManifest.xml`.
- Seule l'activité `MainActivity` est démarrable. Ne mettez pas d'*intent filter* à la nouvelle.

3. Définition des layouts :

- Recopiez le contenu du layout `activity_main.xml` dans `activity_infos.xml`,
- Dans `activity_infos.xml`
  - Changez le titre du message en "Informations",
  - Enlevez tous les boutons sauf le premier,
  - Changez le titre du bouton restant en "Retour".

6.1. Théorie : lancement et terminaison d'une activité

Voici des explications à lire. Il y a un exercice plus loin.

Une activité Android peut lancer une autre activité de la même application, ou d'une autre application. Les activités lancées successivement forment une sorte de pile et seule l'activité du dessus, la plus récente, est visible et active. Quand une activité se termine, soit spontanément, soit avec le bouton *back* ◁, elle est retirée de la pile et c'est l'activité juste en dessous qui redevient visible, et ainsi de suite jusqu'à revenir au bureau Android.

Techniquement, une activité, par exemple `Activity1`, on peut lancer une autre activité, par exemple `Activity2`. La première doit créer un *Intent* et le configurer pour désigner `Activity2` et ensuite démarrer l'activité par l'intent. Voici les instructions :

```
val intentA2 = Intent(this, Activity2::class.java)
startActivity(intentA2)
```

Un *Intent* est un objet qui spécifie le lancement d'une autre activité. Ici, on indique sa classe.

Voici deux particularités de la syntaxe Kotlin (voir fin du TP2) :

1. On ne met pas `new` pour créer une instance. On écrit seulement le nom de la classe et les paramètres du constructeur.
2. Quand veut passer une classe en paramètre, on écrit `Classe::class.java`. Dans la plupart des langages objets, les classes sont elles-mêmes des instances d'une classe spéciale, la classe des classes. L'écriture `Activity2::class.java` récupère l'objet qui représente la classe de `Activity2`.

L'activité qui démarre une autre activité se retrouve comme en dessous de la nouvelle, dans une sorte de pile (*activity stack*). La nouvelle activité occupe tout l'écran avec son layout, et l'ancienne

activité est mise en pause. Quand on appuie sur le bouton *back* ◁, ça termine l'activité du dessus et on revient à l'activité du dessous.

Une activité peut également se supprimer d'elle-même, par exemple quand l'utilisateur a validé un choix, saisi des informations, etc. Il suffit de faire ceci :

```
finish()
```

On peut aussi envisager les instructions suivantes, c'est à dire un lancement suivi d'un `finish()`. L'activité 1 lance la 2 et se termine immédiatement. Et donc elle ne reste pas sur la pile et le le bouton *back* ◁ ne ramènera pas dessus. L'activité 2 remplace entièrement la première.

```
val intentA2 = Intent(this, Activity2::class.java)
startActivity(intentA2)
finish()
```

6.2. Exercice

1. On va ajouter encore deux boutons dans `activity_main.xml` :
   a. Ajoutez un bouton `bouton4` avec le titre "infos+retour",
   b. Ajoutez un bouton `bouton5` avec le titre "infos+fin",
   c. Le premier bouton doit lancer `InfosActivity` en laissant `MainActivity` vivante (ajoutez un écouteur de type référence de méthode en codant ce qu'il faut),
   d. Le deuxième bouton doit lancer `InfosActivity` mais en terminant `MainActivity` (idem).
2. Testez les séquences suivantes (ce sont des tests fonctionnels) :
   - Test 1
     a. Lancez l'application sur un AVD, on doit voir `MainActivity`
     b. Appuyez sur le bouton *back* ◁. Ça doit ramener sur le bureau Android.
   - Test 2
     a. Lancez l'application sur un AVD, on doit voir `MainActivity`
     b. Appuyez sur le bouton 4 "infos+retour", on doit aller sur `InfosActivity`
     c. Appuyez sur le bouton *back* ◁. On doit revenir dans `MainActivity`.
     d. Appuyez sur le bouton *back* ◁. Ça doit ramener sur le bureau Android et non pas dans `InfosActivity`.
   - Test 3
     a. Lancez l'application sur un AVD, on doit voir `MainActivity`
     b. Appuyez sur le bouton 5 "infos+fin", on doit aller sur `InfosActivity`
     c. Appuyez sur le bouton *back* ◁. Ça doit ramener sur le bureau Android et non pas dans `MainActivity`.
3. Maintenant, avec `InfosActivity`, faites en sorte que son bouton "Retour" ramène sur `MainActivity` quand celle-ci est vivante en dessous de `InfosActivity` (lancement avec "infos+retour"), et sur le bureau si ce n'est plus le cas (lancement avec "infos+fin") ?

C'est une question piège. Ce bouton ne doit pas recréer d'*intent* pour démarrer `MainActivity`, mais simplement faire `finish()`. Que se passerait-il si le bouton recréait un *intent* et qu'on re-clique sur le bouton d'infos : la pile d'activité se remplirait d'une alternance de `MainActivity` et `InfosActivity`.

4. Testez les séquences supplémentaires suivantes :
   - Test 4

a. Lancez l'application sur un AVD, on doit voir `MainActivity`
b. Appuyez sur le bouton 4 "infos+retour", on doit aller sur `InfosActivity`
c. Appuyez sur le bouton Retour. On doit revenir dans `MainActivity`.
d. Appuyez sur le bouton *back* ◁. Ça doit ramener sur le bureau Android et non pas dans `InfosActivity`.

- Test 5
  a. Lancez l'application sur un AVD, on doit voir `MainActivity`
  b. Appuyez sur le bouton 5 "infos+fin", on doit aller sur `InfosActivity`
  c. Appuyez sur le bouton Retour. Ça doit ramener sur le bureau Android et non pas dans `MainActivity`.

6.3. Autres types de lancements avec des *Intent*

Le concept d'`Intent` est extrêmement puissant. Cela permet de lancer autre chose que des activités internes d'une application. Par exemple, voici comment lancer un navigateur sur un URL :

```
val url = "https://perso.univ-rennes1.fr/pierre.nerzic/Android"
val intentURL = Intent(Intent.ACTION_VIEW, Uri.parse(url))
startActivity(intentURL)
```

1. Ajoutez un nouveau bouton dans `activity_infos.xml` ayant pour titre "Cours".
2. Définissez un écouteur dans `InfosActivity` qui lance ces instructions (l'*intent* qui ouvre l'URL du cours).

Un *intent* demande une action au système Android. Il y a toutes sortes d'actions possibles, voir la documentation . Dans l'exemple précédent, c'est l'affichage d'un URL. On a aussi des recherches, des éditions, des envois de messages, etc.

Quand on crée un *intent*, on indique le type d'action et on fournit les paramètres nécessaires. Ensuite, le système Android cherche une application capable de traiter cet *intent*. Par exemple, pour une `ACTION_VIEW`, il va chercher un navigateur internet, par exemple Chrome.

Ce sont les applications comme Chrome ou autres qui déclarent dans leur manifeste ce qu'elles sont capables de faire. Par exemple, Chrome déclare être capable d'afficher des URL, un logiciel de mail se déclarera capable d'envoyer des messages et d'en recevoir, un afficheur de documents dira qu'il peut afficher des pdf et des epub, etc.

Quand il y a le choix entre plusieurs applications, le système affiche un dialogue demandant avec quelle application continuer, une seule fois ou toujours...

7. Transmission d'informations entre activités

Maintenant, on va étudier le transfert d'une information entre deux activités. Ça se fait dans l'intent de lancement.

1. Création d'une nouvelle activité :
   - Créez une nouvelle activité de type `Empty Views Activity`,
   - Appelez-la `LoginActivity`. Elle aura un layout appelé `activity_login.xml`.
   - Refaites l'attribution du layout par *view bindings* et l'affichage du titre comme dans le début de `MainActivity.onCreate`.

2. Modification du manifeste :
   - Cette nouvelle activité a été ajoutée dans `AndroidManifest.xml`.
   - Maintenant c'est elle qui doit être la principale. Donc :
     - Déplacez la balise (et son contenu) `<intent-filter>` qui étaient sur `MainActivity` vers `LoginActivity`,
     - Modifiez les attributs `exported="true"` pour celle qui a l'*intent*, `false` pour les autres.
3. Définition du layout :
   - Mettez ce contenu dans `activity_login.xml` :

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical"
    tools:context=".LoginActivity">
    <TextView
        android:id="@+id/message"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Bonjour, qui êtes-vous ?"
        android:layout_marginTop="120dp"
        android:textAppearance="@style/TextAppearance.AppCompat.Large"
        android:gravity="center"/>
    <EditText
        android:id="@+id/username"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:layout_marginTop="20dp"
        android:layout_gravity="center"
        android:hint="Votre pseudo"
        android:inputType="text"
        android:imeOptions="actionDone"/>
</LinearLayout>
```

4. Définition de la méthode `onCreate` et d'un écouteur dans `LoginActivity` :

```
    // interface utilisateur
    lateinit var ui: ActivityLoginBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        // initialisation interne de l'activité
        super.onCreate(savedInstanceState)
        // mise en place du layout par un view binding
        ui = ActivityLoginBinding.inflate(layoutInflater)
        setContentView(ui.root)
        // titre de l'activité
```

```
        setTitle(localClassName)
        // écouteur quand on valide la saisie du pseudo
        ui.username.setOnEditorActionListener(this::onEditorAction)
    }

    private fun onEditorAction(textView: TextView?,
            actionId: Int, keyEvent: KeyEvent?): Boolean
    {
        // récupérer le texte tapé dans l'EditText
        val username = ui.username.getText().toString()
        // TODO lancer MainActivity avec username en tant qu'extra
        ...
        // TODO empêcher un retour dans LoginActivity par le bouton back
        ...
        // pour empêcher un double appel de cet écouteur
        ui.username.setOnEditorActionListener(null)
        return true
    }
```

Il y a deux commentaires marqués `TODO`. Il faut créer un `Intent` pour lancer `MainActivity`, mais en ajoutant ce qu'on appelle des *extras*, c'est à dire des valeurs associées à des clés. Voici les explications.

7.1. Transmission d'extra d'une activité à l'autre

Voici le principe général :

```
    val information: String  = ...
    val nombre: Int = ...
    val intentA2 = Intent(this, Activity2::class.java)
    intentA2.putExtra("information", information)
    intentA2.putExtra("nombre", nombre)     // putExtra a plein de surcharges
    startActivity(intentA2)
```

On ajoute les *extras* dans l'*intent* avant de démarrer l'activité. Il y a de nombreuses surcharges de `putExtra` pour toutes les données possibles – elles doivent être *sérialisables*, c'est à dire qu'il doit y avoir une méthode pour les transformer en chaîne et inversement.

De l'autre côté, dans `Activity2`, il faut récupérer ces *extras* et les extraire dans des variables. Voici le principe :

```
    val information = intent.getStringExtra("information")
    val nombre = intent.getIntExtra("nombre", -1) // il faut une valeur par défaut
```

Attention au piège : le symbole `intent` est un raccourci Kotlin pour `this.getIntent()`. C'est un *getter* pour récupérer l'*intent* qui a servi à lancer cette activité. C'est pour ça que les intents qu'on crée nous-même sont nommés `intentMachin`, pour ne pas les confondre avec le `intent` de l'activité.

Contrairement à `putExtra`, il y a autant de méthodes `get***Extra` que de classes sérialisables, et beaucoup demandent des valeurs par défaut.

Attention, dans le cas d'une chaîne, `getStringExtra` retourne null si l'extra est absent.

On applique cela aux deux activités, `LoginActivity` va envoyer un nom à `MainActivity`.

1. Dans la méthode `onEditorAction` de `LoginActivity`, ajoutez la chaîne `username` en tant qu'*extra* pour lancer `MainActivity`.
2. Dans la méthode `onCreate` de `MainActivity`, ajoutez ce qu'il faut pour récupérer la chaîne `username` présente dans l'*intent*. Puis faites afficher "Bonjour" + ce nom dans le `TextView`.
3. Lancez sur l'AVD. Saisissez votre nom. Pour valider, il faut cliquer sur le bouton représentant un *check* dans le clavier virtuel.

7.2. Amélioration pour éviter des bugs

La solution précédente est assez fragile. Voici une manip à essayer pour s'en rendre compte :

1. Dans `MainActivity.kt`, changez l'instruction
   `val username = intent.getStringExtra("username")` par
   `val username = intent.getStringExtra("usrnam")`.
   C'est une fatue de frappe, et alors ?
2. Lancez l'application sur l'AVD, saisissez votre nom, validez...

Le problème est que l'*extra* qu'on a placé dans l'*intent* est identifié par une chaîne, mais on n'a pas mis la même à l'envoi et à la réception.

Voici comment faire mieux :

1. Dans `MainActivity.kt`, ajoutez ceci au tout début :

```
    companion object {
        // nom de l'extra contenant le nom de connexion
        val EXTRA_USERNAME = "username"
    }
```

Cette syntaxe Kotlin indique de définir une variable de classe appelée `EXTRA_USERNAME`. En Java, on écrirait simplement `public static final String EXTRA_USERNAME = "username";`

2. Dans la méthode `onCreate` de `MainActivity`,
   remplacez `val username = intent.getStringExtra("usrnam")`
   par `val username = intent.getStringExtra(EXTRA_USERNAME)`
3. Dans la méthode `onEditorAction` de `LoginActivity`, remplacez `"username"` par `MainActivity.EXTRA_USERNAME`
4. Vérifiez le bon fonctionnement sur l'AVD.

Les activités sont démarrées à l'aide d'*Intent*. Ce sont des objets qui spécifient une *action* et des paramètres. Parmi les actions, il y a l'affichage de documents. On peut rajouter des informations dans un *intent*, ce qui permet de paramétrer une activité.

8. Travail à rendre

Avec le navigateur de fichiers, ouvrez `~/AndroidStudioProjects/TP3` (comme celui de la première semaine), puis descendez successivement dans `app` puis `src`. Vous devez y voir le dossier `main` ainsi que `test` et `AndroidTest`. Cliquez droit sur le dossier `main` et choisissez `Créer une archive...` puis `.tar.gz` ou `.zip`. Ça doit créer `main.tar.gz` ou `main.zip` contenant tout votre travail.

Rajoutez un fichier appelé exactement `IMPORTANT.txt` dans le sous-dossier `main` si vous avez rencontré des problèmes techniques durant le TP : plantages, erreurs inexplicables, perte du travail, etc. mais pas les problèmes dus à un manque de travail ou de compréhension. Décrivez exactement ce qui s'est passé. Le correcteur pourra lire ce fichier au moment de la notation et compenser votre note. Mettez au moins ce fichier si vous n'avez rien pu faire du tout pendant la séance.

Vous devrez déposer le fichier `main.tar.gz` ou `main.zip` dans le dépôt Moodle Android à la fin du TP, voir sur la page Moodle R4.A11 Développement Mobile .