TP2 - Structure d'une application

Tous les TP se déroulent sur **Windows**, donc redémarrez la machine si c'est Linux.

Cette semaine, on va un peu programmer et continuer à découvrir l'environnement de développement Android. Les actions qu'on va voir se placent plutôt vers la fin d'un projet, pour mettre au point, traduire et distribuer l'application.

1. Affichage d'un message

👉 Créez une nouvelle application type « Empty Views Activity » en Java, nommée TP2.

👉 Ouvrez le source MainActivity.java et complétez-le comme ci-dessous. Ce sont des instructionspour afficher un message dans la fenêtre LogCat :

```
import ...
import android.util.Log      // ou taper ALT-entrée pour l'ajouter auto

    public class MainActivity extends AppCompatActivity {
        private final String TAG = "TP2";
        @Override
        protected void onCreate(Bundle
    savedInstanceState) {
            super.onCreate(savedInstanceState);
            setContentView(R.layout.activity_main);
            Log.i(TAG, "Mon premier message de log !");    }
    }
```

NB: la syntaxe Kotlin sera expliquée au fur et à mesure des besoins. Pour l'instant, on est d'accord, ça paraît bizarre.

👉 Ouvrez l'onglet LogCat en bas de l'écran, figure 1.

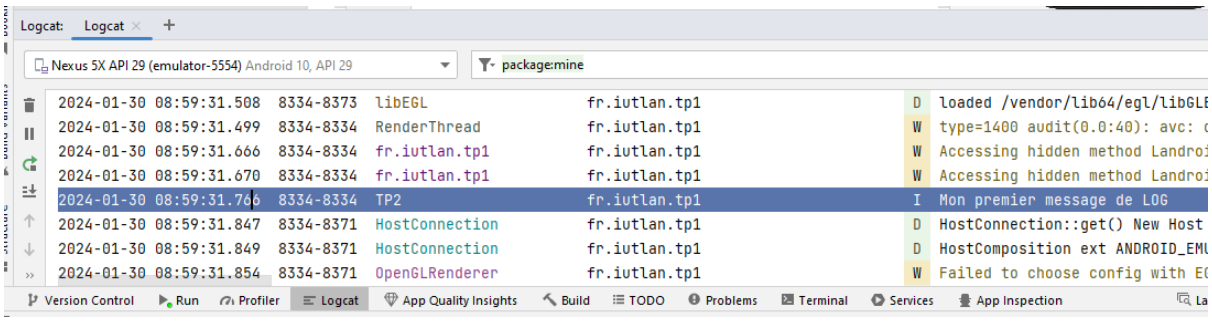

Figure 1: LogCat

Cette fenêtre permet de voir tous les messages émis par la tablette.

Lancez l'exécution du projet. Vous devriez voir cette ligne, mais totalement noyée dans le reste :

```
01-27 09:58:38.310 992-992/ mr.supnum.tp2 I/TP2: Mon premier message de log !
```

Il y a la datation du message, le numéro de processus, le *package* de votre application. Ensuite la lettre I correspondant à la gravité du message, puis l'étiquette TAG, suivis du message.

Il y a plusieurs niveaux de gravité (ou *sévérité*) pour les messages :

- D (*debug*) pour des messages de mise au point, savoir que telle méthode est appelée
- I (*info*) pour des messages généraux, des commentaires sur ce qui se passe
- W (*warning*) pour des messages d'erreurs récupérables
- E (*error*) pour des messages d'erreurs graves

L'intérêt, c'est de pouvoir masquer les messages d'un niveau inférieur. Par exemple, on peut choisir de n'afficher que les messages de niveau W et E. Cela se fait en configurant un *filtre*. On peut demander à n'afficher que les messages venant d'un certain *package*, portant une certaine étiquette *TAG*, et d'une gravité au moins égale à un niveau.

On peut également ne vouloir voir que les messages portant un TAG précis. Pour tout cela, on définit un filtre dans le haut de la fenêtre.

👉 Complétez le filtre en package:mine tag=:TP2 level:INFO. Le système vous aide à compléter chaque item.

NB: parfois le LogCat se déconnecte de l'application. On ne voit plus arriver les messages de Log attendus. Il faut re-sélectionner l'émulateur dans la liste déroulante.

1.1. Calcul d'une racine carrée par la méthode de Héron

👉 On va maintenant ajouter une méthode à la classe MainActivity :

```
/**
 * calcul d'une racine carrée par la méthode de Héron
 *
 * @param N nombre dont on veut la racine carrée
 * @return racine carrée de N
 */
public static double racine2(double N) {
    double n = N / 2;
    for (int i = 1; i <= 50; i++) {
        double m = N / n;   // donc la surface du rectangle (n, m) est N = n
* m

        n = (n + m) / 2;    // n = moyenne entre n et m
        if (Math.abs(n - m) < 1e-12) {
            break;
        }   // arrêt si pas d'amélioration
```

```
    }
    return n;
}
```

```
public static void testRacine2() {
    double N = 2.0;
    while (N < 50.0) {
        double n = racine2(N);
        System.out.println("racine(" + N + ")  =  " + n);

        N += 7.4;
    }
}
```

👉 Ajoutez ces instructions dans onCreate (gardez ce qui existe) :

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);


    Log.i(TAG, "Mon premier message de log !");
    testRacine2();
}
```

L'affichage peut être amélioré à l'aide d'un *template* (patron ou modèle). On retrouve ce concept en JavaScript avec les chaînes `...`.

👉 Effectuez la transformation suivante dans testRacine2 :

```
Log.i(TAG, String.format("racine(%f) = %f", N, n));
```

Pour info, la méthode de Héron d'Alexandrie calcule une racine carrée par une méthode intéressante, voir Wikipedia. C'est une méthode géométrique, très visuelle. On part d'une valeur n plus petite que N , par exemple $n = N/2$. Ensuite on construit un rectangle dont la surface vaut N , avec l'un des côtés de longueur n. Il faut que l'autre côté soit de longueur N/n, ainsi la surface est $n \times N/n = N$. Ensuite, la méthode consiste à faire en sorte que les deux côtés n et N/n se rapprochent peu à peu, de manière à ce que le rectangle devienne progressivement un carré. Quand les deux côtés n et N/n sont identiques, le rectangle est devenu un carré, et n est alors la racine carrée de N . Alors pour faire se rapprocher les côtés, la méthode consiste à remplacer n par la moyenne entre n et N/n. C'est ce qui est codé ci-dessus.

Dans le programme ci-dessus, il y a exactement 50 tours de boucle. Il serait préférable d'arrêter dès que n et m sont plus proches qu'un certain *delta* tout petit. C'est parce que cette méthode est

très efficace. Le nombre de décimales exactes double à chaque tour de boucle, et donc 50 itérations sont trop nombreuses.

👉 Ajoutez cette ligne après le calcul de *m* dans la boucle :

```
if (Math.abs(n - m) < 1e-12) {
    break;
}   // arrêt si pas d'amélioration
```

👉 Pour la fonction abs, il y a le choix entre Math.abs et la bibliothèque math de java. Si vous choisissez la seconde, il faudra importer java.math.abs.

Notez au passage comment on compare deux réels, par la valeur absolue de leur différence.

👉 Consultez la documentation sur les structures de contrôle. On n'aura pas besoin de grand chose pour les prochains TP, mais il faut savoir faire une conditionnelle et des boucles *tant que* et *pour*.

2. Mise au point

Nous allons expérimenter quelques aspects du débogueur intégré dans Android Studio. On va découvrir quelques techniques très utiles : points d'arrêt et exécution pas à pas.

Il va falloir modifier deux fichiers qui sont liés. On commence par le *layout*.

👉 Remplacez tout le fichier res/layout/activity_main.xml par ceci (passez en mode Code pour voir le XML). Ça sera expliqué en détails la semaine prochaine :

```
<?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="wrap_content"
    android:layout_gravity="center"
    tools:context=".MainActivity"
    android:orientation="vertical">

    <TextView
        android:id="@+id/textview"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:gravity="center"/>

    <Button
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Permuter"
        android:onClick="onPermuter"/>
</LinearLayout>
```

👉 Complétez le source de MainActivity avec ceci (conservez ce qui est déjà là) :

```java
package com.example.tp2;

import androidx.appcompat.app.AppCompatActivity;

import android.os.Bundle;
import android.util.Log;
import android.view.View;
import android.widget.TextView;

import java.util.Arrays;

public class MainActivity extends AppCompatActivity {

    private static final String TAG = "TP2";
    private TextView textView;
    private final int[] tableau = {1, 3, 6, 8, 9};


    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);


        Log.i(TAG, "Mon premier message de log !");
        double n = racine2(49.0);
        Log.i(TAG, "racine(49) = " + n);
        textView = findViewById(R.id.textview);
        display(tableau);
        //testRacine2();
    }

    private void display(int[] tab) {
        textView.setText(Arrays.toString(tab));
    }

    /** appelée quand on clique sur le bouton, voir TP4 */
    public void onPermuter(View view) {
        Log.i(TAG, "clic sur le bouton Permuter");
        permutation(tableau);
        display(tableau);
    }

    /**
     * fait une permutation circulaire croissante des valeurs :
     * [a, b, c, d] => [d, a, b, c]
     * @param tab tableau à permuter
     */
    private void permutationa(int[] tab) {
    int dernier = tab[tab.length - 1]; // dernier élément
    for (int i = 0; i < tab.length - 1; i++) {
        recopierSurSuivant(tab, i);
```

```
    }
    tab[0] = dernier;
}



}
```

👉 Réglez LogCat pour n'afficher que des messages du niveau warn ou error, venant de votre *package*, mais de n'importe quelle étiquette : package:mine level:warn.

Le but du programme est de faire tourner les valeurs du tableau : [1, 2, 3, 4] doit devenir [4, 1, 2, 3].

2.1. Lancement normal

👉 Appuyez sur l'icône poubelle du LogCat pour effacer les précédents messages.

👉 Lancez l'application sur un AVD. Cliquez sur le bouton.

2.2. Exécution en pas à pas

Ensuite, on voit que le programme ne fonctionne pas bien. Il ne plante plus mais ne semble pas faire ce qui est attendu : permuter les valeurs affichées. Faites comme si vous ne compreniez pas son fonctionnement. On va regarder vivre le programme en traçant son exécution.

Le principe est de placer des *points d'arrêt* (*break points*). Ce sont des marques ajoutées à certaines instructions qui indiquent au système de s'arrêter là. On retrouve la main, on peut examiner les valeurs des variables et éventuellement relancer l'exécution jusqu'au prochain point d'arrêt.

Il faut comprendre que les arrêts se font selon ce qui est exécuté. Si vous placez un point d'arrêt dans une méthode qui n'est jamais appelée…

4.2.1. Placement d'un point d'arrêt

👉 Cliquez dans la marge gauche au début de la méthode onCreate(). Ça place un gros point rouge dans la marge, figure 2.

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);


    Log.i(TAG, "Mon premier message de log !");
}
```

Figure 2: Point d'arrêt dans le source

N'en mettez pas trop dans un programme, vous aurez du mal à suivre tout ce qui peut se passer. Mettez vos méthodes au point les unes après les autres, pas tout en même temps.

Pour commencer, on met un point d'arrêt dans la méthode `onCreate`, on regarde ce qui se passe là, déjà et ensuite, on déplace le point d'arrêt là où il semble que l'exécution dévie de ce qui est attendu.

4.2.2. Lancement en mode debug

👉 Cliquez sur le bouton entouré en bleu pour lancer en mode mise au point, figure 3.

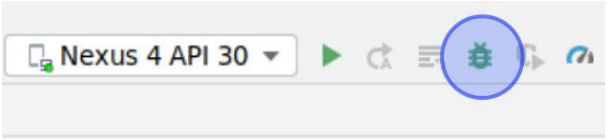

Figure 3: Bouton de mise au point

Comme vous avez mis un point d'arrêt dans `onCreate`, l'éxecution s'est arrêtée là. La ligne est colorée en bleu foncé. Android Studio affiche la pile d'exécution à ce stade, figure 4.

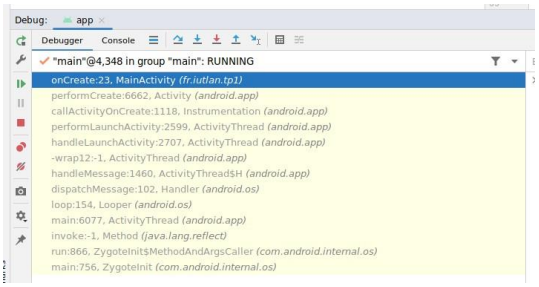

Figure 4: Pile d'exécution

Voici, figure 5, les boutons utiles pour mettre le programme au point. Le bouton entouré en vert *resume program* permet de faire repartir le programme jusqu'au prochain point d'arrêt. Le bouton bleu clair *step over* permet d'exécuter la prochaine ligne sans rentrer dans les méthodes imbriquées s'il y en a. Le bouton orange *step into* entre dans les appels imbriqués de méthodes. Le bouton violet *step out* exécute à pleine vitesse jusqu'à la sortie de la fonction puis s'arrête. Le bouton entouré en bleu foncé affiche les points d'arrêt. Nous n'emploierons pas les autres boutons.

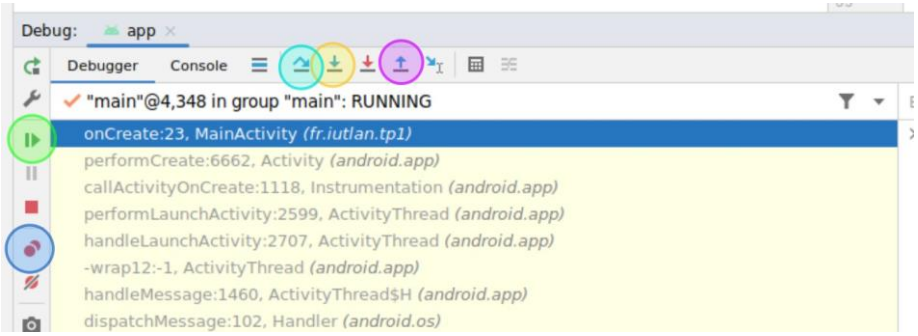

Figure 5: Bouton d'exécution pas à pas

La fenêtre `Breakpoints`, qu'on obtient en cliquant sur le bouton entouré en bleu foncé dans la figure précédente, liste tous les points d'arrêts que vous avez placés.

👉 Appuyez sur le bouton bleu clair *step over*. L'exécution du `super.onCreate()` a été faite. Appuyez à nouveau sur *step over* (raccourci `F8`) pour exécuter `setContentView()`. En fait, rien ne se passe sur l'écran. Appuyez à nouveau sur *step over* jusqu'à arriver sur la fin de la méthode, l'accolade fermante, mais pas plus loin (car vous allez tomber dans les classes compilées Android). On n'a pas vu grand chose d'intéressant mais au moins, ça ne s'est pas planté.

👉 Enlevez le point d'arrêt de `onCreate` et mettez-en un sur la première instruction de la méthode `permutation`, puis cliquez sur le bouton *resume program*. L'exécution reprend à vitesse normale.

👉 Cliquez sur le bouton `Permuter` dans l'AVD. Ça va activer la méthode `onPermuter` qui appelle `permutation`, là où il y a le point d'arrêt.

4.2.3. Examen des variables

D'abord, on va regarder le contenu des variables. Regardez la pile d'exécution. Les deux niveaux du haut sont `permutation:50` et `onPermuter:39`. Cliquez sur chacun, cela affiche toutes les données associées : paramètres et variables locales.

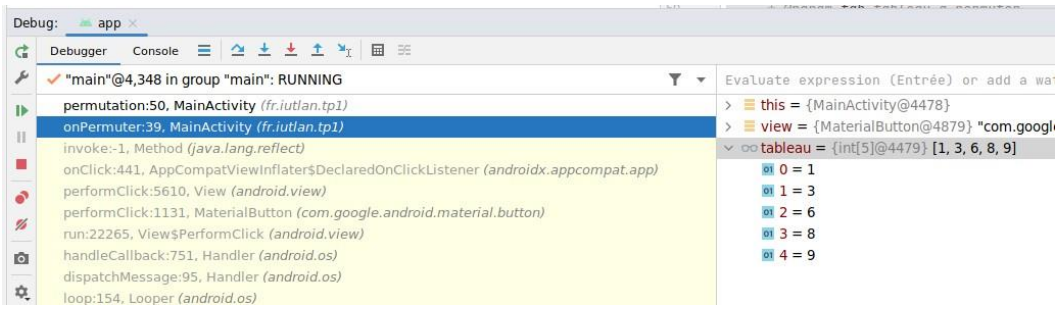

Figure 6: Variables de chaque niveau de la pile

On s'intéresse à la variable `tableau` de `onPermuter`. On va la *surveiller*, c'est à dire suivre toutes ses modifications.

👉 Dans le source, cliquez droit sur la variable `tableau` et sélectionnez `Add to Watches`, figure 7. Ça va la rajouter dans la liste des variables toujours affichées à droite, avec une paire de lunettes pour indiquer que vous la surveillez. Si vous changez de *frame* (niveau de pile), elle sera encore affichée.

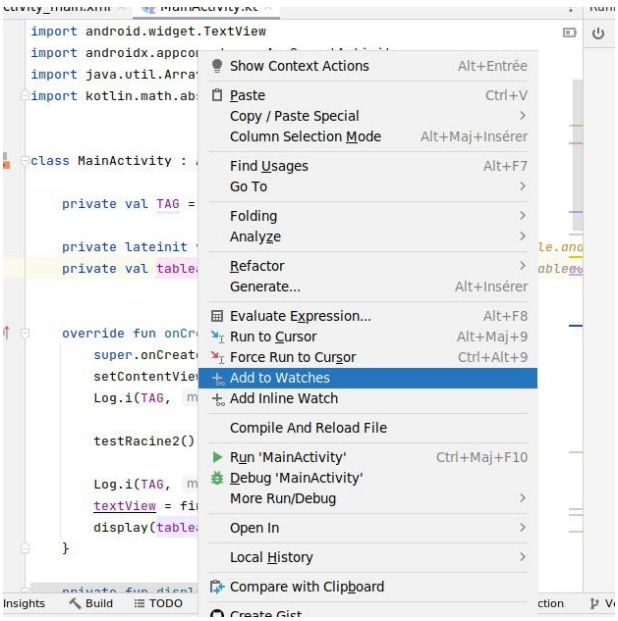

Figure 7: Ajout aux variables surveillées

👉 Dépliez-la dans la fenêtre Debug. Comme c'est un petit tableau, ses valeurs sont toutes visibles. Déjà, on constate que l'ordre de ces valeurs est le même que sur l'écran de l'AVD, donc le bug ne vient probablement pas de l'affichage (il se pourrait que la permutation marche bien, mais que ça soit à l'affichage que ça déraille…).

👉 Maintenant, appuyez sur *step over* ou F8 en surveillant les valeurs de tableau et de l'indice i de la boucle. On voit que la valeur 1 de la première case est recopiée sur toutes les cases suivantes.

👉 Relancer l'exécution du début (bouton bestiole puis cliquer sur le bouton permuter dans l'AVD pour arriver au point d'arrêt).

👉 Maintenant, appuyez sur *step into* ou F7. Cette fois, l'exécution suit vraiment tous les appels de méthode. On voit vraiment ce qui ne va pas dans le programme.

3. Création d'un APK distribuable

Si vous voulez distribuer votre application, il faut en faire un *apk* (Android Package). C'est l'équivalent d'un *jar* qui contient les binaires et les ressources, prêt à être installé.

C'est très simple : menu Build, item Build Bundle(s) / APK(s), puis Build APK(s). C'est tout. Vous trouverez le fichier en question dans
C:\...\AndroidStudioProjects\TP2\app\build\intermediates\apk\debug

Ce ne sera que la version de mise au point. Elle n'a pas les qualités pour être réellement publiée. Elle contient par exemple les liens entre instructions et lignes de programme. D'autre part, cet apk n'est pas épuré et obscurci (*obfuscated*) pour être moins facilement décorticable. Pour finir, cet *apk* n'est pas signé, et donc son authenticité pourrait être compromise par des gens malveillants.

Android Studio propose de construire des variantes *build variants*, dont la distribuable Release, mais c'est un peu plus compliqué. Il faut notamment signer le fichier numériquement. Voici les étapes.

1. Menu Build, item Generate Signed Bundle/APK
2. Dans la boîte de dialogue figure 8, choisissez APK. L'autre option, Android App Bundle, est « *un format de publication qui contient l'ensemble du code et des ressources compilés de votre application, et qui délègue la génération de l'APK et sa signature à Google Play* » (doc Android).

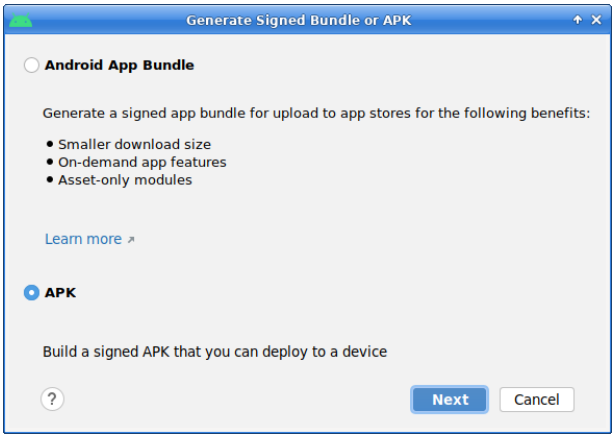

Figure 8: Choisir APK

3. Dans la suite de la boîte, figure 9, AS demande l'emplacement de la boîte à clés numériques. Vous n'en avez pas encore, alors cliquez sur le bouton Create new....

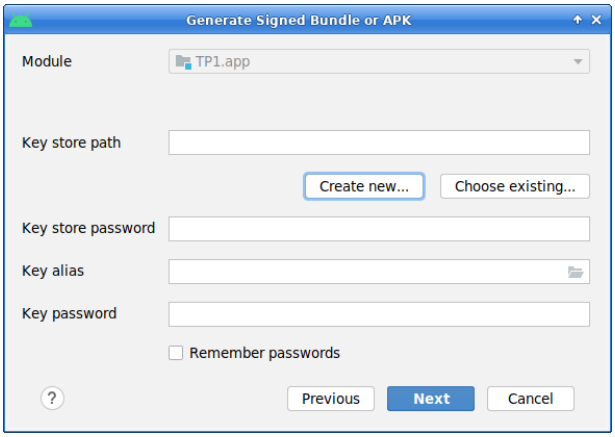

Figure 9: Créer trousseau de clés

4. Il s'agit de créer un trousseau de clés et une clé, figure 10. N'oubliez pas de mettre tous les mots de passe (mettez votre prénom pour faciliter ce TP). Validez quand c'est fini.

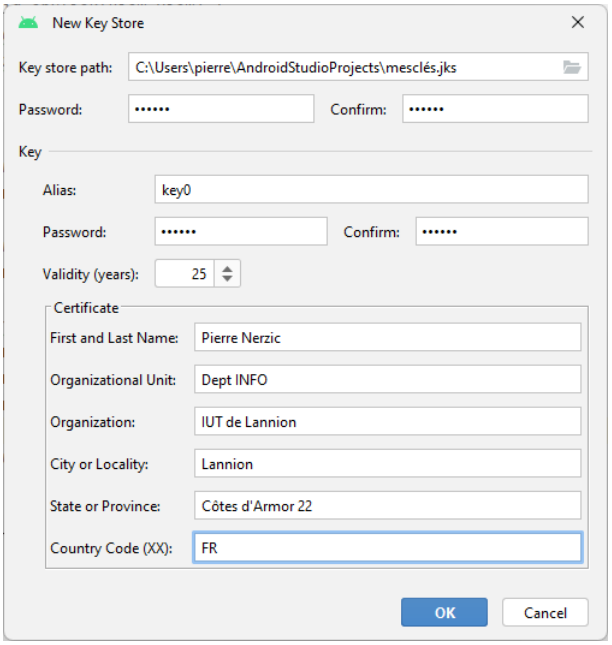

Figure 10: Définition du trousseau de clés

5. On revient dans le dialogue de choix d'une clé. Les champs sont maintenant remplis, figure 11. Il se peut qu'il y ait un souci avec le chemin du trousseau de clés. Il est préférable que ce chemin soit absolu. Cliquez sur Choose existing... si ce n'est pas le cas et localisez le trousseau. Remettez les mots de passe. Cochez Remember passwords.

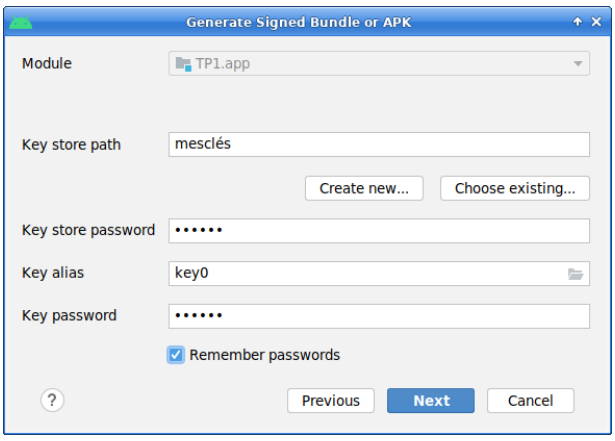

Figure 11: Trousseau de clés créé

6. Il reste à générer l'*apk* signé en version release (cliquez dessus), figure 12. Vous le trouverez dans le dossier app\release de votre projet.

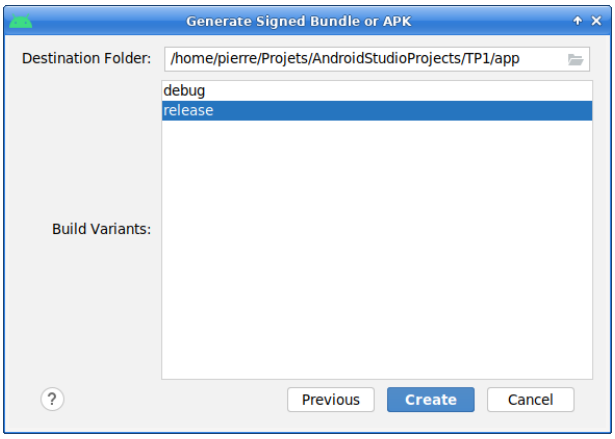

Figure 12: Génération de l'APK

6. Traduction d'une application

Ces manipulations vont nous faire découvrir la puissance des ressources dans Android. Les ressources sont des fichiers associés aux sources, comme des icônes, des dispositions d'interfaces (*layouts*), et des traductions.

Par exemple, il y a un texte, « Permuter » sur le bouton rajouté précédemment qu'il faudrait traduire en anglais si on voulait vendre l'application. Actuellement ce texte se trouve en dur, dans le fichier `res/layout/activity_main.xml`. Si vous passez la souris au dessus de ce texte, vous aurez une suggestion pour le placer dans les ressources.

6.1. Mettre un texte dans les ressources

1. Double-cliquez sur `res/layout/activity_main.xml`, il est peut-être déjà ouvert dans un onglet.
2. Affichez le contenu XML de ce fichier en cliquant sur `Code`
3. Il est probable que la ligne `android:text="Permuter"` soit colorée en orange, signe qu'on peut faire mieux. Placez le curseur dessus. Normalement une ampoule apparaît peu après.
4. Cliquez sur l'ampoule et ensuite `Extract string resource`

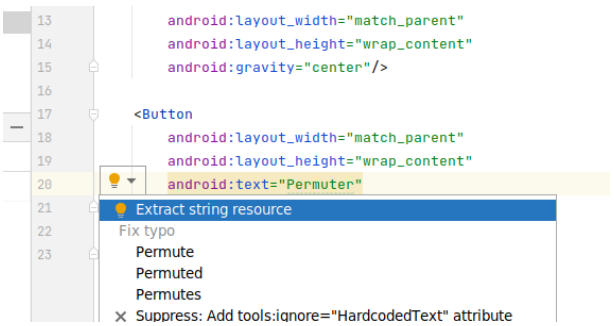

Figure 13: Menu quick fix

5. Le dialogue suivant définit un identifiant pour la chaîne (*Resource name*). En principe, on choisit un identifiant international et facile à comprendre pour des traducteurs.

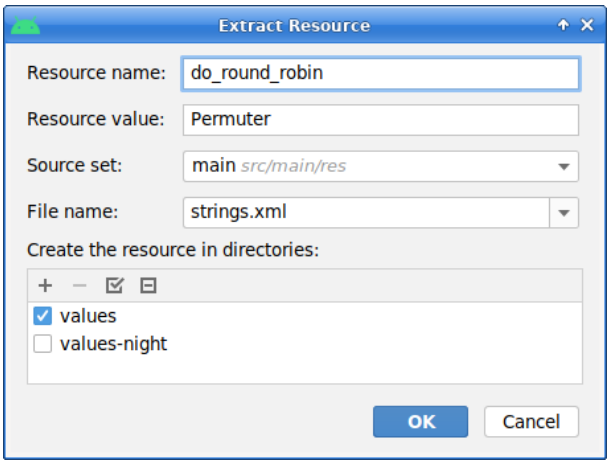

Figure 14: Dialogue d'extraction

C'est fini. Deux choses se sont passées :

a. `android:text="Permuter"` a été remplacé par `android:text="@string/do_round_robin"`. C'est ce qu'on appelle une référence de ressource.
b. Dans le fichier `res/values/strings.xml`, il y a cette nouvelle balise.

```
<string name="do_round_robin">Permuter</string>
```

En résumé, placer un texte dans les ressources signifie :

- ajouter une balise `<string name="un_identifiant">texte</string>` dans `res/values/strings.xml`
- remplacer le texte en dur (*hard coded*) par une référence `<Button ... android:text="@string/un_identifiant" ...`

La syntaxe `@string/identifiant` est une référence à une ressource de type chaîne, placée dans `res/values/strings.xml`.

6.2. Traduire les ressources

Cela consiste à faire des variantes du fichier `strings.xml`, une pour chaque langue visée. Ces variantes seront dans différents dossiers, par exemple `res/values-fr/strings.xml`, `res/values-de/strings.xml`, `res/values-en/strings.xml`, `res/values-es/strings.xml`, etc. Chacun de ces fichiers contiendra les mêmes balises avec les mêmes identifiants que `res/values/strings.xml`, mais les textes seront traduits.

Android Studio facilite ce travail.

1. Double-cliquez sur le fichier `res/values/strings.xml` pour l'afficher dans un éditeur.
2. Cliquez sur les mots `Open editor` en haut à droite de la zone où il y a le contenu XML. C'est un éditeur de traductions.

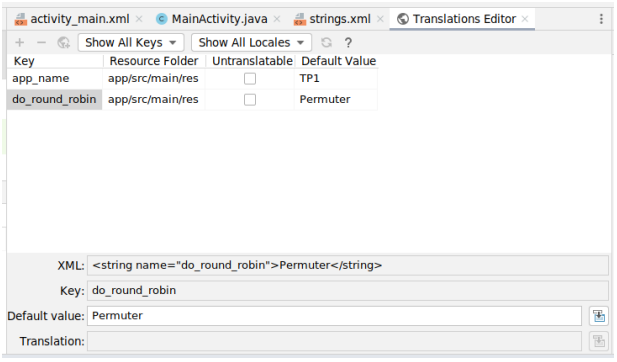

Figure 15: Éditeur de traductions

3. Cliquez sur le bouton en forme de globe terrestre en haut à gauche, et ajoutez une « Locale ». Cherchez French (fr) dans la liste.
4. Une nouvelle colonne est apparue. Mettez Permuter pour le français et Round Robin pour la valeur par défaut. Cochez Untranslatable pour TP2.
5. Maintenant, il faut lancer l'application sur l'AVD. Changez les paramètres linguistiques, anglais <-> français pour voir le changement.

Énormément de choses peuvent être reconfigurées en fonction des paramètres du smartphone, message, dispositions, icônes, thèmes, etc.

.