Cette semaine, on va un peu programmer et continuer à découvrir l'environnement de développement et la structure d'une application Android.

1. À chaque séance

Vérifiez que vos quotas disque ne sont pas épuisés. Si c'est le cas, tout fichier que vous créez directement (sources) ou indirectement (compilation ou autre) sera vide. Inutile de détailler les conséquences ?

2. Affichage d'un message

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

Si vous avez créé une application de type « Empty Activity » avec le logo Compose, une sorte de cube, vous aurez seulement à supprimer cette application et la recréer correctement. On ne peut pas passer d'un type à l'autre à cause de toutes les dépendances qui se mettent en place.

👉 Complétez le source `MainActivity.kt` comme ci-dessous, là où c'est fléché `ICI`. Ce sont des instructions pour afficher un message dans la fenêtre `LogCat` :

```
import ...
import android.util.Log         // <---- ICI

class MainActivity : AppCompatActivity() {

    private val TAG = "TP2"     // <---- ICI

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        enableEdgeToEdge()
        setContentView(R.layout.activity_main)
        ViewCompat.... (à ignorer) {
            ... (à ignorer) ...
            insets
        }

        // message dans le log
        Log.i(TAG, "Mon premier message de log !") // <---- ICI
    }
```

NB: la syntaxe Kotlin sera expliquée au fur et à mesure des besoins.

👉 Ouvrez le panneau `LogCat` en cliquant sur l'icône en forme de tête de chat entouré en vert, figure 1.

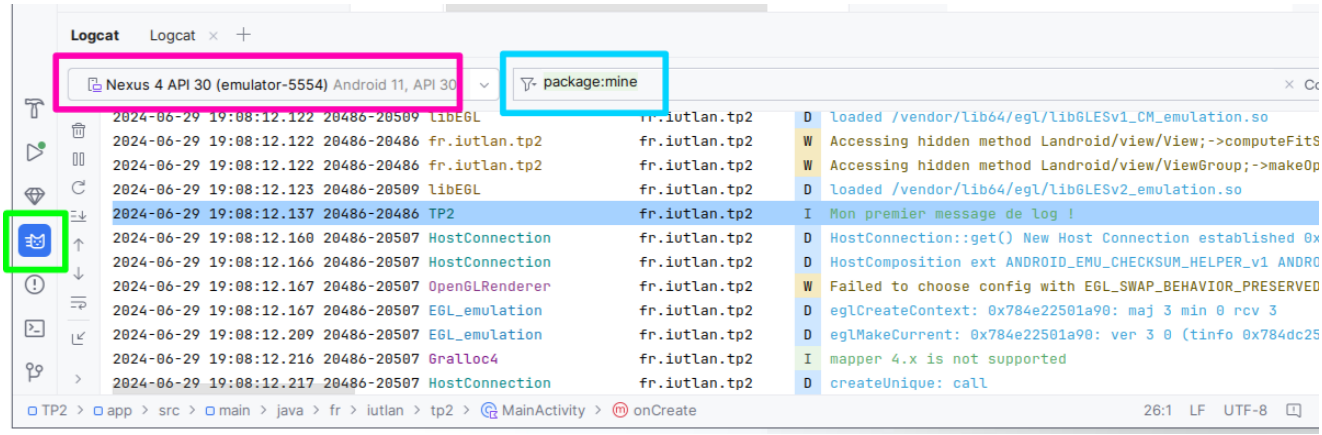

Figure 1: LogCat

Cette fenêtre permet de voir tous les messages émis par la tablette. C'est le service *syslog* de son système Linux.

☛ Lancez l'exécution du projet. Vous devriez voir cette ligne, sélectionnée dans la figure 1, mais totalement noyée dans le reste :

```
2024-06-29 19:08:12.137 20486 TP2  fr.iutlan.tp2  I  Mon premier message de log !
```

Il y a la datation du message, le numéro de processus (PID), l'étiquette TAG, le *package* de votre application, ensuite la lettre `I` correspondant à la gravité du message, suivis du message.

Il y a plusieurs niveaux de gravité (ou *sévérité*) pour les messages. Les voici par gravité croissante :

- `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, zone entourée en bleu.

☛ Complétez le filtre en `package:mine tag=:TP2 level:INFO`. Le système vous aide à compléter chaque item. Vous n'aurez plus que les messages que vous avez programmés.

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, zone entourée en rouge/rose.

C'est l'instruction `Log.i(TAG, message)` qui sert à émettre un message au niveau *info*. Pour les autres niveaux, il y a `Log.d`, `Log.w` et `Log.e`. Vous pouvez choisir le `TAG` comme vous voulez, pour pouvoir le trouver facilement dans le LogCat.

Il est indispensable de savoir écrire des messages dans le LogCat. C'est la première chose qu'on

vous demandera quand vous demanderez de l'aide pour comprendre pourquoi une application ne fonctionne pas : avez-vous fait afficher des messages pour savoir ce qui se passe ou pas ?

3. Explications sur Kotlin

On reprend le source `MainActivity.kt`. Il commence comme un source Java, par le *package*.

```
package fr.iutlan.tp2
```

On ne met pas de point virgule en fin de ligne, sauf si on veut mettre plusieurs instructions à la suite.

```
import android.util.Log
```

Ce sont les mêmes importations qu'en Java. Ici, ce sont celles du SDK Android et vous les apprendrez au fil de l'eau.

3.1. Définition d'une classe simple

En Kotlin, on définit une classe par `class NomClasse : SuperClasse {`

```
class MainActivity : AppCompatActivity() {
```

En Java, on écrirait `class NomClasse extends SuperClasse {`

Il y a un petit point à remarquer, c'est que, ici, la superclasse `AppCompatActivity` possède un constructeur, qu'il faut appeler. Ça veut dire qu'en Java, on aurait ceci :

```
public class MainActivity extends AppCompatActivity
{
    MainActivity() {
       super();
       //... autres initialisations ...
   }
```

Sauf, qu'en Java, quand il n'y a pas d'autres initialisations, il n'y a pas besoin d'écrire le constructeur, il est implicite. Mais en Kotlin, on doit mettre des parenthèses après la superclasse pour dire qu'on appelle son constructeur : `class MainActivity : AppCompatActivity() {`.

👉 Essayez d'enlever ces parenthèses : `class MainActivity : AppCompatActivity {`. Le message est *This type (AppCompatActivity) has a constructor, and thus must be initialized here.* Ça veut dire qu'il faut appeler le constructeur de `AppCompatActivity`, et en Kotlin, on doit mettre des parenthèses.

👉 Faites un CTRL-clic gauche sur `AppCompatActivity`. Vous arrivez sur le code source et vous voyez que c'est du Java. Kotlin et Java sont compilés dans le même langage machine appelé « bytecode ART ». En principe, on peut utiliser une classe Java dans du Kotlin. C'est le cas ici, mais pas toujours parce qu'il y a plusieurs versions de bytecode.

On revient au fait de devoir mettre des parenthèses à la superclasse. Le constructeur de `AppCompatActivity` appelle lui-même `super()` puisque c'est du Java et qu'il y a une autre méthode à appeler, `initDelegate`, peu importe ce qu'elle fait. Donc il faut que le constructeur

de `MainActivity` appelle le constructeur sans paramètres de `AppCompatActivity`. C'est ce que signifient ces parenthèses.

Une autre solution consiste à ajouter un constructeur explicite à la classe `MainActivity`, et dans ce cas, vous enlevez les `()` après le nom de la superclasse et voici comment faire :

```
class MainActivity : AppCompatActivity {

    constructor() {
        //... autres initialisations ...
        Log.i(TAG, "constructeur de MainActivity")
    }
```

En Java, le constructeur porte le nom de la classe, tandis qu'en Kotlin comme en JS, c'est le mot clé `constructor`.

À noter que l'emploi de ce mot clé `constructor` permet de définir un *constructeur secondaire*, et ça ne plait pas à Android Studio : `Secondary constructor should be converted to a primary one`. C'est parce que, normalement, on définit d'abord un *constructeur primaire* dans la première ligne `class`. Cela se ferait ainsi, mais ça n'a aucun intérêt ici :

```
class MainActivity : AppCompatActivity() {

    init {
        //... autres initialisations ...
        Log.i(TAG, "constructeur de MainActivity")
    }
```

Ça commence à être sensiblement différent du Java, et ce n'est que le début.

3.2. Définition de variables membres

Les variables membres se définissent ainsi en Kotlin :

```
val titre: String = "le titre"        // JAVA: final String titre = "le titre";
var message: String = "le message"    // JAVA: String message = "le message";
```

On met `val` pour une constante (*read-only*), `var` pour une variable (*mutable*).

Le type peut être omis s'il est sans ambiguïté. Par exemple :

```
var nombre = 3      // attention, nombre est un Int
var coef = 3.0      // coef est un Double
```

Attention, parce que le type est fixé à la compilation, en fonction de la valeur initiale. On ne peut plus le changer dynamiquement à chaque affectation, comme en JavaScript.

Un aspect intéressant de Kotlin apparaît quand il n'y a pas de valeur initiale. Voici un exemple :

```
var nom: String         // INTERDIT, ne se compile pas car pas initialisé
```

Une telle variable vaudrait `null` en Java, et l'utiliser sans précaution peut déclencher une exception. Kotlin interdit cela. Vous devez l'initialiser dans *tous* les constructeurs, donc éventuellement rajouter un constructeur s'il n'y en a pas.

Ou alors mettez le mot clé `lateinit` devant la variable pour dire que vous l'initialiserez ultérieurement :

```
lateinit var nom: String    // Ok, mais n'oubliez pas d'initialiser avant usage
```

Mais très bizarrement, Kotlin est ensuite incapable de détecter que l'initialisation n'a pas été faite, et ça entraîne une exception à l'exécution, très dommage... Kotlin propose seulement ce test :

```
if (this::nom.isInitialized) {  /* ok pour utiliser nom dans ce bloc */ }
```

Comme en Java, on peut modifier la visibilité des variables membres.

```
public var nombre = 0
private var coef = 3.0
protected lateinit var prenom: String
```

On reparlera des variables membres et des classes dans un prochain TP.

3.3. Définition d'une méthode

On va maintenant ajouter une méthode à la classe `MainActivity`.

👉 Insérez ce code après la méthode `onCreate` :

```
/** calcul d'une racine carrée par la méthode de Héron
 * @param N nombre dont on veut la racine carrée
 * @return racine carrée de N
 */
fun racine2(N: Double): Double {
    var n = N / 2
    for (i in 1..50) {
        val m = N / n       // donc la surface du rectangle (n, m) est N = n * m
        n = (n + m) / 2     // n = moyenne entre n et m
    }
    return n
}

fun testRacine2() {
    var N = 2.0
    while (N < 50.0) {
        val n = racine2(N)
        Log.i(TAG, "racine("+N+") = "+n)
        N += 7.4
    }
}
```

Il y a plusieurs différences avec Java.

- Les paramètres et types du résultats sont mis après les noms.
- Quand il n'y a pas de résultat, on n'écrit pas `void`, on ne met rien.

- Les variables locales sont définies comme les variables membres, `val` ou `var` selon qu'elles sont constantes ou non.
- Une boucle *pour* bornée (nombre d'itération fixé à la compilation) est très différente en Kotlin. Consultez cette page .
- la notation `n1..n2` construit une *étendue* (*range*). C'est un itérateur sur les valeurs allant de $n1$ à $n2$. Consulter cette page pour approfondir.

👉 Ajoutez ces instructions dans `onCreate` en gardant l'existant :

```
override fun onCreate(savedInstanceState: Bundle?) {
    ...
    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, "racine($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 autant d'itérations sont inutiles.

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

```
    if (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 Kotlin. Si vous choisissez la seconde, il faudra importer `kotlin.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*.

4. Interface utilisateur minimale

Nous allons améliorer ce petit logiciel. Au lieu de voir le résultat, la racine carrée dans le LogCat, on va l'afficher sur l'écran, dans une interface utilisateur, avec de quoi saisir une valeur en entrée

du calcul et un bouton pour lancer le calcul.

Les interfaces utilisateur dans Android se programment de deux manières totalement incompatibles :

- Une manière traditionnelle basée sur des composants d'affichage appelés *vues* (*Views*), configurées par des fichiers XML appelés *layouts* (dispositions). Créer une interface, c'est créer un *layout* XML en positionnant différentes vues les unes par rapport aux autres. Ça ressemble énormément à une page HTML, comme des `<button>` dans des `<div>` avec des attributs pour les styles, sauf que ce sont des balises et attributs totalement différents. Ensuite, on programme les écouteurs des vues sous forme de méthodes dans la classe `ActivityMain`.
  Il faut encore connaître cette manière de faire, car il reste encore beaucoup d'applications faites comme ça.
- Une nouvelle manière (depuis juillet 2021) appelée Jetpack Compose. On verra cette API à partir du TP4, pas avant car elle est assez complexe, demande une bonne compréhension de Kotlin et de la structure d'une application.

Donc, dans ce TP2, on va mettre en place une interface à l'aide d'un *layout* et ensuite on le rendra vivant avec les écouteurs.

4.1. Mise en place de l'interface

4.1.1. Spécification XML

👉 Dans le panneau du projet à gauche, dépliez les « ressources », c'est à dire l'item `res` en dessous de `kotlin+java`, puis la catégorie `layout` et vous allez voir `activity_main.xml`. Voir la figure 2.

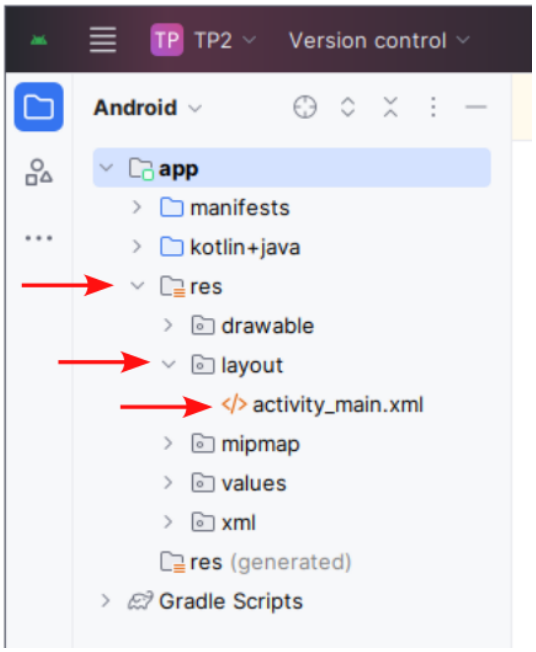

Figure 2: Panneau projet

Cet item correspond à un dossier qui se trouve dans `AndroidStudioProjects/TP2/app/src/main/res`. Le fichier `activity_main.xml` se trouve dans le sous-dossier `layout`.

👉 Utilisez le navigateur de fichiers de Linux pour aller dans `res` puis `layout`.

👉 Ouvrez le fichier `activity_main.xml` dans un éditeur de texte simple comme geany, gedit ou autre.

Voici le début de ce fichier :

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintLayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    ...
    tools:context=".MainActivity">

    <TextView ... />

</androidx.constraintLayout>
```

C'est un fichier XML. XML et HTML sont deux frères ayant le même ancêtre, un langage appelé SGML. Il y aura un cours et un TP sur ce sujet dans la matière R4.02. Dans HTML, les balises sont destinées à de la mise en page, tandis qu'en XML, les balises peuvent être quelconques, choisies par la norme du fichier, pour représenter des données complexes. Ici, ce sont des balises pour définir des composants d'interface, `ConstraintLayout` et `TextView`, et ces balises sont spécifiées par les attributs `xmlns` de la balise racine.

Ici, chaque élément XML, `<ConstraintLayout>` et `<TextView>` décrit un élément qu'on verra sur l'interface quand le logiciel sera lancé.

AndroidStudio possède un éditeur graphique pour les *layouts*.

👉 Ouvrez le même *layout* dans AndroidStudio. Vous allez voir un atelier d'édition, voir figure 3.

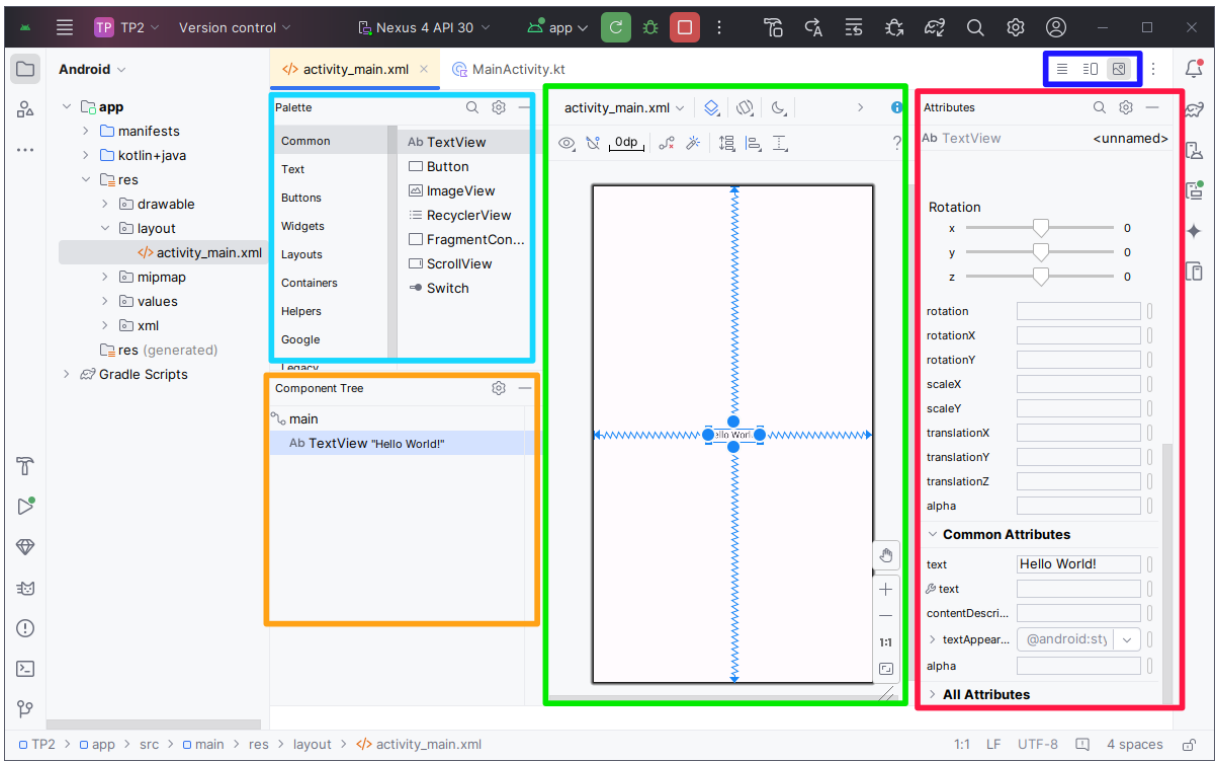

Figure 3: Éditeur de layout

Il y a plusieurs zones et boutons à connaître :

1. Entouré en vert, un aperçu de la future interface. On peut choisir l'orientation portrait/paysage du téléphone.
2. Entouré en orange, l'arbre des vues de l'interface. Le composant racine `<ConstraintLayout>` contient une vue enfant `<TextView>`.
3. Entouré en rouge, les propriétés, c'est à dire la configuration, de l'élément sélectionné.
4. Entouré en bleu clair, un panneau avec tous les composants visuels utilisables. On peut les faire glisser sur l'écran, mais cela implique de savoir exactement ce qu'on fait, notamment vis à vis du placement sur l'écran pour que ça soit agréable sur tous les smartphones.
5. Entouré en bleu foncé, trois boutons pour choisir le mode de travail : XML, graphique ou mixte.

Il faut bien comprendre que cet éditeur visualise et modifie le XML sous-jacent. Tout dans l'interface utilisateur se retrouve écrit d'une certaine manière dans le XML. Les composants visuels sont sous forme d'éléments XML et leur configuration est sous forme d'attributs. Nous verrons ultérieurement ce qu'il en est des icônes, titres et autres. Quand on veut créer une application ainsi, il faut bien connaître les vues et leurs propriétés. Il faut du temps pour ça.

Cet éditeur graphique permet de construire rapidement une interface, en glissant déposant des éléments de la palette. Par contre la mise en page ne fonctionnera pas bien sur tous les téléphones et tablettes parce qu'il faut ajuster le placement et les dimensions d'une manière précise qu'on ne peut pas faire avec la souris. Ça se fait en choisissant les bonnes propriétés et les bonnes valeurs dans les attributs des éléments XML. Donc ce qui est plutôt recommandé quand on maîtrise Android, c'est de travailler en mode mixte : le XML à gauche et la prévisualisation à droite.

👉 Cliquez sur le `TextView` au centre de l'interface dans la zone verte, ou bien dans la zone orange,

faites défiler les propriétés dans la zone rouge et modifiez celle qui s'appelle `text`. Mettez un autre texte comme `TP2` ou `Mon appli`. Ça modifie l'aperçu en temps réel. Sauvez le fichier. Constatez que ça a changé l'attribut `android:text` dans le fichier XML (éditeur texte ou mode XML dans l'éditeur zone bleu foncé).

👉 Relancez l'application. Le message a maintenant changé sur l'écran. De temps en temps, Google propose une fonctionnalité de mise à jour en temps réel sur l'AVD, mais ça ne marche pas toujours bien.

👉 Cherchez la propriété `textAppearance` un peu en dessous de `text` et commencez à taper `Large` en tant que valeur. Ça va vous proposer toutes les valeurs connues contenant le mot `Large`. Sélectionnez `@style/TextAppearance.AppCompat.Large`. Le texte sera un peu plus visible.

Vous pouvez un peu explorer toutes les propriétés proposées.

4.1.2. Changement de *layout*

Le *layout* d'origine basé sur un `ConstraintLayout` est trop complexe à modifier pour ce premier vrai TP. On va le remplacer par beaucoup plus simple.

👉 Passez en mode `Code` zone bleu foncé pour voir le XML puis remplacez tout le contenu par celui de tp2_activity_main.xml (le fichier est `tp2_activity_main.txt` à renommer en `activity_main.xml` si vous le téléchargez ; c'est un `.txt` pour s'afficher correctement si vous l'ouvrez dans un onglet du navigateur).

Le composant `<LinearLayout>` est destiné à aligner ses composants enfants soit verticalement, soit horizontalement, selon son attribut `android:orientation`. Il y en a deux dans cette interface, regardez comment ils sont imbriqués.

Les attributs `android:layout_width` et `android:layout_height` sont obligatoires. Ils définissent la taille des vues. Il y a trois valeurs autorisées : `match_parent` = la vue s'allonge pour remplir toute la place libre, `wrap_content` = la vue demande la place minimum, `0dp` = quand il y a aussi `android:layout_weight` pour définir une sorte de pourcentage de cette vue par rapport aux autres. Ici, le bouton a un poids de 0, mais une largeur de `wrap_content`, donc il fera sa largeur minimale et le edittext prendra toute la place restante. Voici un exemple d'attributs qu'on place bien plus rapidement à la main qu'en essayant de le faire avec l'interface.

Toujours dans l'élément `<EditText>`, son attribut `android:inputType` définit le type de contenu qu'on peut saisir, des nombres. Il y a d'autres possibilités qui changent le clavier virtuel proposé à l'utilisateur.

Nous n'aurons pas le temps de creuser davantage la spécification d'interfaces de ce genre. Nous passerons plus de temps à étudier Jetpack Compose qui est plus moderne mais très différent.

4.2. Écouteurs

Si vous lancez l'application, vous verrez que tout est opérationnel, mais que le bouton ne fait rien. Nous allons lui associer une méthode.

4.2.1. Définition de la méthode

👉 Voici ce qu'il faut rajouter dans la classe `MainActivity` :

```
fun onCalculer(view: View) {
    try {
        // trouver le EditText dans lequel on tape le nombre
        val tvNombre = findViewById<EditText>(R.id.nombre)

        // extraire la chaîne tapée
        val txtNombre = tvNombre.text.toString()

        // analyser ce qui a été tapé en tant que nombre
        val nombre = txtNombre.toDouble()

        // calculer sa racine
        val racine = racine2(nombre)

        // trouver le TextView dans lequel on affiche le résultat
        val tvResultat = findViewById<TextView>(R.id.resultat)

        // afficher le résultat
        tvResultat.text = racine.toString()
    } catch (e: Exception) {
        Log.e(TAG, e.toString())
    }
}
```

Il va y avoir quelques importations à faire : `View`, `EditText` et `TextView`. Vous n'avez qu'à taper alt+entrée pour valider les propositions.

Il y a quelques points intéressants dans cette fonction (méthode).

- La fonction `findViewById<EditText>(identifiant)` cherche le `<EditText>` dans l'interface qui possède un attribut `android:id="@+id/nombre"`. Allez voir lequel c'est. On retrouve la même fonction plus loin pour affecter `tvResultat`.

  Dans un layout, on ajoute un attribut `android:id="@+id/IDENTIFIANT"` à une vue pour définir son identifiant unique. Et dans le source kotlin, on utilise `R.id.IDENTIFIANT` pour désigner cet identifiant.

- L'interface est mise en place par l'instruction `setContentView(R.layout.activity_main)` qui se trouve dans la fonction `onCreate` (allez voir). Là aussi, `R.layout.NOMLAYOUT` est une sorte d'identifiant. Il désigne le fichier XML entier.

  Le symbole `R` est une classe statique qui contient tous les identifiants et noms des ressources. Ils sont regroupés dans différentes catégories, `R.id` pour les identifiants des vues, `R.layout` pour les layouts, `R.string` pour les chaînes, `R.drawable` pour les icônes, etc. On les découvrira au fur et à mesure.

- Quand on récupère une vue avec `findViewById`, on la place dans une variable et ensuite on peut manipuler la vue avec ses setters et getters. En Kotlin, les setters et getters ressemblent à des variables membres. Voyez `tvResultat.text`. C'est la même chose que `android:text` dans le layout. C'est le texte affiché. On affecte `tvResultat.text` pour le modifier. En réalité, c'est vraiment un setter. Passez la souris dessus pour voir la Javadoc. Control-cliquez

dessus pour aller sur la définition de ce setter. Lui aussi est en Java.

4.2.2. Association de l'écouteur au bouton

👉 Ajoutez l'attribut `android:onClick="onCalculer"` à l'élément `<Button>` du layout.

Cet attribut associe la méthode indiquée à l'action sur le bouton. Cette méthode doit avoir un seul paramètre, de type `View`, la superclasse de tous les composants d'interface et ne doit rien retourner.

👉 Lancez l'application sur un AVD. Maintenant tout doit fonctionner.

5. Traduction d'une application

On voudrait vendre cette application, TP2, dans le monde entier, afin que tout le monde puisse bénéficier de racines carrées fraîches à toutes les heures de la journée. On va être obligé de la traduire dans différentes langues parce que les gens ne connaissent pas le français. En anglais, on appelle ça la *localization* de l'application.

Ces manipulations vont nous faire découvrir la puissance des ressources dans Android. Les ressources sont des fichiers associés aux sources, comme des icônes, les interfaces, et aussi les traductions des messages.

Par exemple, il y a un texte, « Calculer » sur le bouton et il faudrait le traduire. Actuellement ce texte est codé en dur (*hardcoded*), dans le fichier `res/layout/activity_main.xml`.

5.1. 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. Regardez le premier `<TextView>`, le `<EditText>` et le `<Button>`. Ils ont un attribut `tools:ignore`. Il sert à masquer les avertissements. Supprimez cet attribut en totalité.
4. Maintenant tous les attributs `android:text` et `android:hint` sont colorés en orange. Placez le curseur dessus. Normalement une information apparaît peu après : `Hardcoded string "quel nombre ?", should use @string resource`.

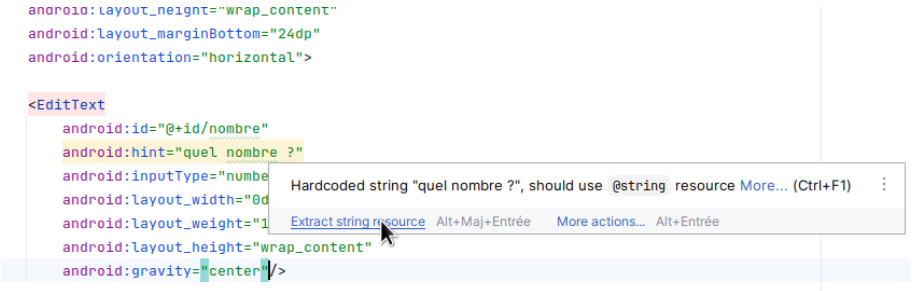

Figure 4: Menu quick fix

5. Cliquez sur le lien `Extract string resource`

6. 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. Il faut que les traducteurs sachent exactement comment traduire ces mots.

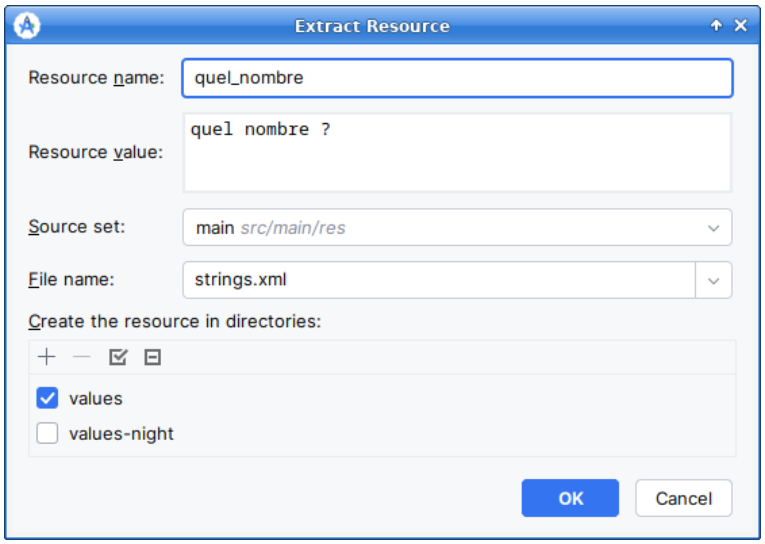

Figure 5: Dialogue d'extraction

C'est fini. Deux choses se sont passées :

a. `android:hint="quel nombre ?"` a été remplacé par `android:hint="@string/quel_nombre"`. C'est ce qu'on appelle une référence de ressource.
Vous pouvez cliquer sur cette référence. Ça ouvrira le fichier `res/values/strings.xml`.
b. Dans le fichier `res/values/strings.xml`, il y a cette nouvelle balise.

```
<string name="quel_nombre">quel nombre ?</string>
```

La chaîne a été déplacée dans le fichier `strings.xml` et remplacée dans le layout par une référence.

En résumé, placer un texte dans les ressources signifie :

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

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

5.2. 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.

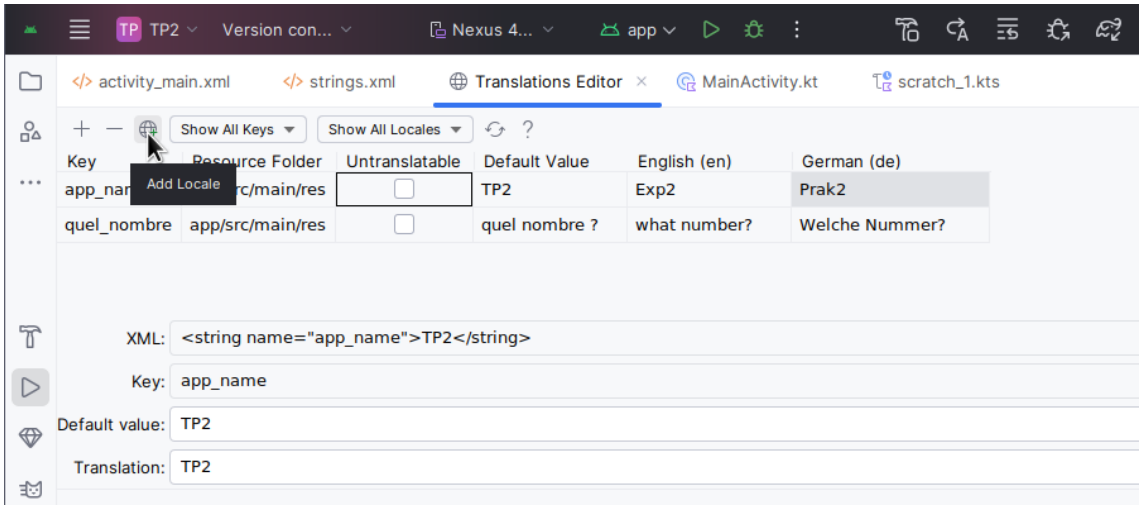

Figure 6: Éditeur de traductions

3. Cliquez sur le bouton en forme de globe terrestre en haut à gauche, et ajoutez une « Locale ». Cherchez la langue que vous voulez dans la liste, `French (fr)` par exemple. Il va y avoir une langue par défaut, en général l'anglais et des langues alternatives. Dans la copie écran, c'est le français par défaut.
4. Une nouvelle colonne est apparue. Mettez `quel nombre ?` pour le français et `what number?` pour la valeur par défaut. Vous pouvez coche `Untranslatable` pour « TP2 », le nom de l'application ou bien chercher comment ça se traduit.
5. Maintenant, il faut lancer l'application sur l'AVD. Changez les paramètres linguistiques dans les préférences, anglais <-> français pour voir le changement.

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

6. Liaison entre interface et programme

Nous avons vu comment une méthode peut accéder aux vues de l'interface, en les cherchant avec `findViewById`. Cette méthode n'est plus recommandée car elle est lente quand l'interface contient beaucoup de vues. Et aussi, comme elle est basée sur la recherche d'identifiants, il se peut que la vue voulue ne soit pas dans le *layout* installé, mais dans un autre.

On va donc creuser la question de la mise en place d'un *layout* et de la manipulation de ses vues.

6.1. Situation actuelle

La spécification de l'interface, le *layout*, est définie dans un fichier XML. On peut le nommer comme on veut, du moment qu'il se trouve dans `res/layout`.

L'association de ce layout à l'activité `MainActivity` est faite dans sa méthode `onCreate`. Voici sa structure schématique, si on enlève tout ce qui est inutile aux explications :

```
class MainActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        // initialisation de l'activité
        super.onCreate(savedInstanceState)

        // mise en place de l'interface
        setContentView(R.layout.activity_main)

        ...
    }
}
```

La méthode `setContentView` définit le layout à utiliser dans cette activité. Remarquez l'écriture `R.layout.activity_main`. C'est une référence au fichier `res/layout/activity_main.xml`. Le nom du fichier se transforme en identifiant.

Ensuite, dans les méthodes, on peut récupérer une vue `<TYPE android:id="IDENTIFIANT".../>` de ce layout par `findViewById<TYPE>(R.id.IDENTIFIANT)`.

6.2. Passage aux *View Bindings*

Depuis quelques années, il est très recommandé d'utiliser une autre méthode appelée *View Bindings*. Ça consiste à ce que Android Studio crée automatiquement une classe nommée un peu comme le layout et contenant des variables membres typées comme les vues de ce layout et nommées d'après leurs identifiants.

Par exemple avec le layout `activity_main.xml` du TP, on va obtenir une classe appelée `ActivityMainBinding` contenant deux variables membres, `nombre` de type `EditText` et `resultat` de type `TextView`. Les vues qui n'ont pas d'identifiant ne sont pas mises dans cette classe.

Mettre en place un layout consiste alors à instancier cette classe et l'associer à l'activité. Ensuite, on se sert directement des variables membres.

6.2.1. Fichier `build.gradle.kts` de l'application

La génération des *View Bindings* n'est pas mise en place dans les projets de base proposés par Android Studio. Il faut configurer le projet. Cela se fait en modifiant l'équivalent du fichier `pom.xml`, sauf qu'avec Android, c'est l'un des deux fichiers `build.gradle.kts`.

👉 Dépliez la catégorie `Gradle Scripts` dans le panneau du projet, en dessous des ressources.

👉 Constatez qu'il y a deux fichiers `build.gradle.kts`, le premier pour le projet TP2 entier, et le second pour le module `app`. On voit ces deux fichiers dans la figure 7, le premier en vert clair, le second en orange.

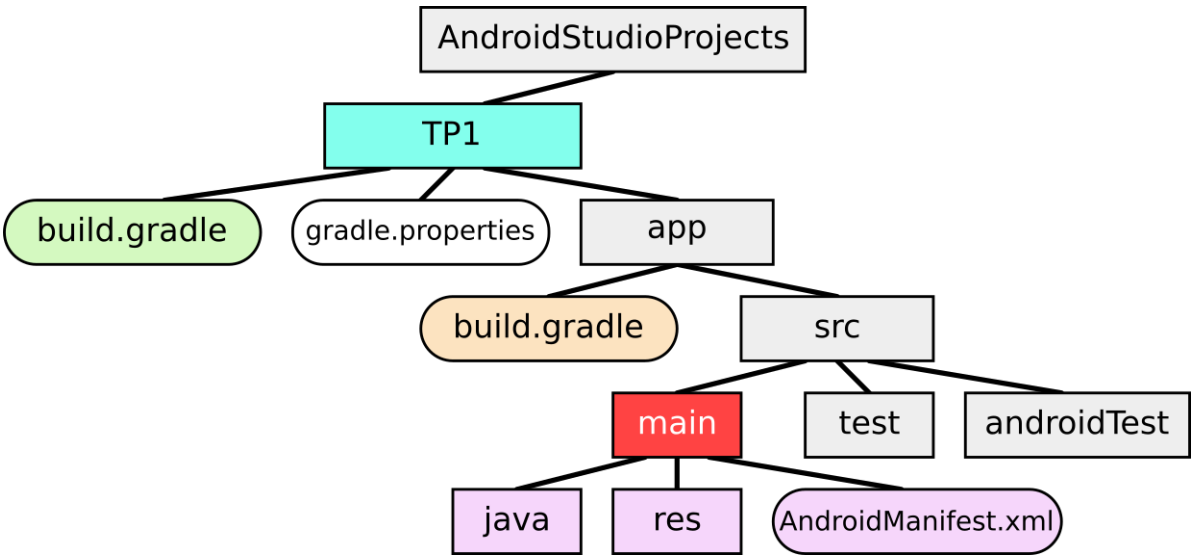

Figure 7: Structure des fichiers d'un projet Android

C'est le second qui nous intéresse, celui qui est dans le dossier `app`. Ouvrez-le et remarquez la structure en blocs `buildTypes {...}`, `compileOptions {...}` etc. Il faut juste rajouter trois lignes au même niveau que ces blocs, comme dans la figure 8.

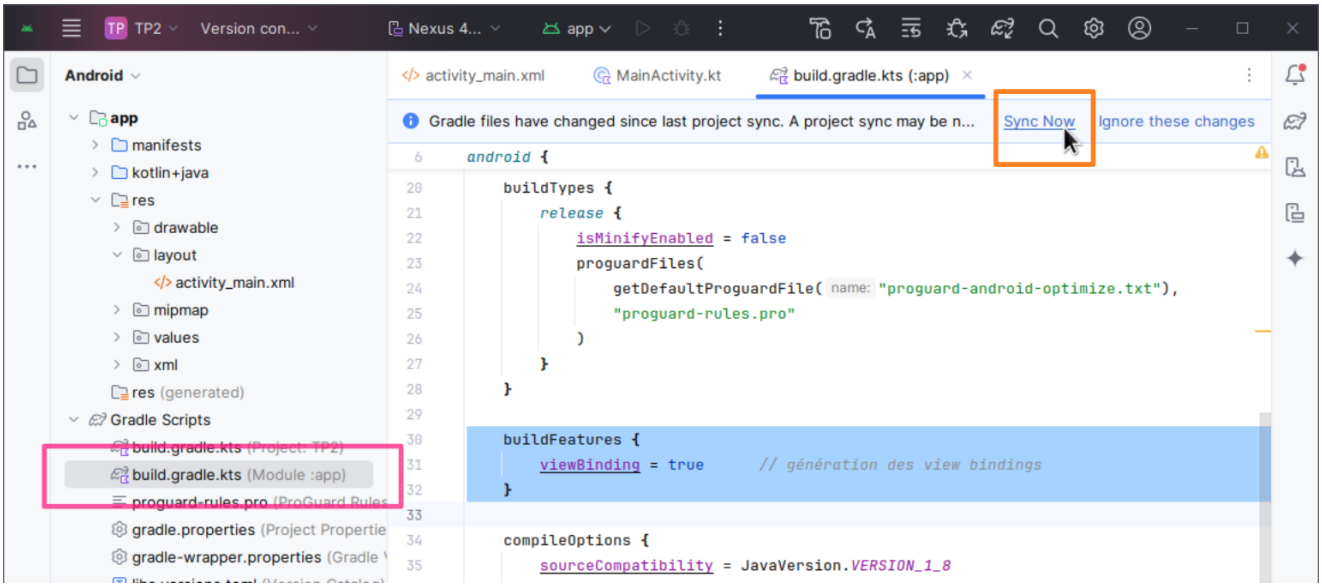

Figure 8: build.gradle.kts du module app

Voici les trois lignes à rajouter au bon endroit :

```
    buildFeatures {
        viewBinding = true      // génération des view bindings
    }
```

👉 Cliquez sur le bouton `Sync Now` qui est apparu en haut, à chaque changement dans ce fichier. Ça va reconstruire le projet entier.

6.2.2. Mise en place de l'interface

Maintenant, la classe `ActivityMainBinding` est automatiquement générée par Android Studio. Il reste à créer l'interface. Voici le principe (à ne pas copier/coller sans réfléchir attentivement) :

```
class MainActivity : AppCompatActivity() {

    lateinit var ui: ActivityMainBinding

    override fun onCreate(savedInstanceState: Bundle?) {
        // initialisation de l'activité
        super.onCreate(savedInstanceState)

        // mise en place de l'interface
        ui = ActivityMainBinding.inflate(layoutInflater)
        setContentView(ui.root)

        ...
    }
}
```

👉 Faites le changement, en préservant les autres instructions qui sont déjà présentes, sauf l'ancien appel à `setContentView`. Il ne doit rester qu'un seul appel à `setContentView`, celui avec `ui.root` en paramètre.

6.2.3. Accès aux vues

Maintenant, la variable `ui` donne un accès direct aux vues de l'interface.

👉 Mettez-vous devant la méthode `onCalculer`. Elle contient deux structures `val VUE = findViewById<...>(R.id.IDENTIFIANT)` suivie d'une utilisation de `VUE.text`.

👉 Remplacez tous les `VUE.text` par `ui.IDENTIFIANT.text`. Par exemple, `tvNombre.text` devient `ui.nombre.text`.

👉 Supprimez les lignes comme `val ... = findViewById<...>(R.id.IDENTIFIANT)`

La méthode devient un peu plus courte et surtout beaucoup plus fiable. Il n'est pas possible de se tromper à cause d'un mauvais identifiant ou d'un mauvais type, car la compilation serait bloquée.

7. Classes Kotlin

On termine par quelques explications sur la programmation objet en Kotlin. À continuer sur la prochaine séance, si ce n'est pas terminé.

👉 Pour essayer les extraits qui suivent, ouvrez un fichier « scratch » : menu `File`, item `New`, sous-item `Scratch File`. Ensuite, tapez « ko » et choisissez `Kotlin` (tout court). Ça ouvre une fenêtre assez géniale, vous tapez ou modifiez du code source Kotlin à gauche et ça affiche les résultats à droite, quasiment en temps réel.

NB: Si vous voyez `...` en vert très clair, cliquez dessus pour afficher tout le texte.

La version complète se trouve dans tp2_kotlin_scratch.txt .

7.1. Définition d'une classe

Voici une classe toute simple :

```
class Animal(val nom: String, var mois: Int) {

    fun saluer(mot: String = "Bonjour", fin: String = ".") {
        println("$mot $nom$fin")
    }

    override fun toString(): String {
        return "$nom : $mois mois"
    }
}
```

Kotlin est extrêmement compact comparé à Java.

1. La première ligne déclare à la fois une classe, son constructeur, ainsi que deux variables membres. En Kotlin, si on ajoute des sortes de paramètres préfixés par `val` ou `var` après le nom de la classe, ils deviennent des constantes ou variables membres.
2. Il y a une méthode `saluer` qui prend un paramètre optionnel. Cette valeur est utilisée si le paramètre `mot` n'est pas fourni à l'appel.
3. Il y a une surcharge de la méthode `toString()`.

7.2. Instancier une classe

Pour créer une instance, on ne met pas `new`. Ajoutez ceci après la classe :

```
val monchien = Animal("filou", 36)

monchien.saluer()
println("Je possède ${monchien}.")
```

L'appel à la méthode `saluer` se fait sans arguments, et donc ce sont les valeurs par défaut qui sont utilisées.

La dernière instruction fait afficher : `Je possède filou : 36 mois.` car il y a un appel implicite à la méthode `toString()`. Dans un modèle de chaînes (*template*), on peut entourer une variable par des `{}` pour la séparer du reste, mais ce n'est pas obligatoire.

Les arguments des fonctions et constructeurs peuvent être nommés et dans ce cas, on peut les fournir dans un autre ordre (NB: dans ce qui suit, les ... signifient de garder ce qu'il y avait déjà). Ajoutez ceci à la suite :

```
val monchien2 = Animal(mois=64, nom="toufou")

monchien2.saluer(fin=" !", mot="Coucou")
println("Je possède ${monchien2}.")
```

7.3. Initialisation

Dans certains cas, on a besoin d'initialiser l'instance en cours de création. Modifiez la classe comme ceci :

```
class Animal(val nom: String, var mois: Int) {
    ...

    init {
        println("ouaf miaou cuicui")
    }
}
```

C'est un peu comme un complément au constructeur. Toutes les instances créées afficheront ce message. Normalement, on initialise des variables membres.

7.4. Types simples

Les types de base sont un peu différents de ceux du Java. Ce sont : `Boolean`, `Char`, `String`, `Int`, `Float`... Il n'y a pas la distinction que fait Java entre `int` et `Integer` par exemple.

7.5. Variables membres

Quand on définit une variable membre dans une classe, Kotlin crée automatiquement un *setter* et un *getter*. Ajoutez ceci à la fin :

```
monchien.mois = 38
println("Je possède ${monchien.nom} qui a ${monchien.mois} mois")
```

Les *getters* et *setters* sont écrits comme si c'étaient de simples variables membres publiques en Java, mais en interne, ce sont bien des méthodes, générées automatiquement, qui sont appelées. C'est assez compliqué et voici davantage d'explications.

En Java, toute variable membre est stockée en mémoire.

```
/* ne pas essayer dans le scratch, regardez seulement ce que c'est */
class Voiture {
    private String marque;
    private int annee;

    public Voiture(String marque, int annee) {
        this.marque = marque;
        this.annee = annee;
    }
}

Voiture macaisse = new Voiture("citrengeot", 2010)
```

Dans la mémoire, il y a des octets réservés pour chacune des variables membres.

Kotlin a une vision un peu différente. Une classe contient des *propriétés* (*properties*) qui peuvent ne pas être stockées en mémoire. Une propriété Kotlin est constituée d'un *setter* et/ou *getter*.

Voici l'exemple de propriétés supplémentaires qui ne sont pas stockées en mémoire, mais qui sont utilisables exactement comme des variables membres. Complétez la classe comme ceci :

```
class Animal(val nom: String, var mois: Int) {
    ...

    var annees      // type Int optionnel, déduit du getter
        get() = mois / 12
        set(annees) {
            mois = annees * 12
        }

    val isJeune: Boolean
        get() {
            return mois < 36
        }
}
```

Ajoutez ceci à la fin, pour tester :

```
monchien.annees = 13        // setter sur une propriété non stockée en mémoire
println("${monchien.nom} a ${monchien.mois} mois, càd ${monchien.annees} ans.")
println("${monchien.nom} est-il jeune ? ${monchien.isJeune}.")
```

Les propriétés `annees` et `isJeune` ne sont que calculées. Elles ne sont pas stockées en mémoire. Pour cela, il faut programmer à la fois le *setter* et le *getter*.

De plus, la propriété `isJeune` est déclarée `val`, car elle n'est pas modifiable (dans ce cas, pas de *setter*).

S'il y a une vraie variable membre occupant des octets en mémoire, comme `nom` et `mois`, alors cette variable est appelée *champ* (*field*). Ce champ est qualifié de *backing field* (doublage de la propriété par de la mémoire). Pour définir un champ, on fait ainsi (complétez la classe) :

```
class Animal(val nom: String, var mois: Int) {
    ...

    var dateNaissance: String? = null
        // getter et setter générés automatiquement
}
```

Ajoutez ceci à la fin, pour tester :

```
monchien.dateNaissance = "31/02/2019"
println("Il est né le ${monchien.dateNaissance}, je crois...")
```

Dans un *setter*, pour affecter le champ en mémoire, on ne peut pas utiliser `this.champ` parce que ça appellerait le setter récursivement. Il faut employer le mot-clé `field`. Voici une propriété avec champ, *getter* et *setter* :

```
class Animal(val nom: String, var mois: Int) {
    ...
```

```
    var race: String = "?"
        get() {
            return "${field} très joueur"
        }
        set(t) {
            field = "sublime $t"
        }
}
```

Ajoutez ceci à la fin, pour tester :

```
monchien.race = "épagneul breton"
println("C'est un ${monchien.race}.")
```

Le `println` affiche `"C'est un sublime épagneul breton très joueur."`. Notez que ce *setter* enregistre une valeur modifiée, et ce *getter* re-modifie la valeur qui avait été enregistrée – normalement, on ne fait pas ça.

Il y a plusieurs syntaxes pour les *getter*, soit `get() = expression`, soit `get() { return expression }`.

7.6. Héritage

Pour définir une sous-classe, il faut que la superclasse soit dérivable. Pour cela, il faut que le mot-clé `open` lui ait été ajouté lors de sa définition. Ajoutez ce mot-clé et ajoutez ensuite la classe `Chat` et les instructions de test :

```
open class Animal(val nom: String, var mois: Int) {
    ...
}

class Chat(nom: String, val couleur: String, mois: Int) : Animal(nom, mois) {
    override fun toString(): String {
        return "chat $nom : $couleur, $mois mois"
    }
}

val monchat = Chat("dikie", "beige et blanc", 96)
monchat.saluer()
println("Je possède ${monchat} et ${monchien}.")
```

On ne doit pas mettre `var` ou `val` pour des propriétés qui sont dans la superclasse, uniquement pour celles qui sont dans la sous-classe.

7.7. Méthode et constantes de classe

En Java, les méthodes et variables qui s'appliquent à la classe sont définies avec le mot-clé `static`. On les utilise ensuite par `NomClasse.nomMéthode(...)` et `NomClasse.nomVariable`, c'est à dire sur la classe elle-même. En Kotlin, la définition est syntaxiquement assez bizarre. On définit un objet qui accompagne la classe. Mais ensuite, à l'usage, c'est comme en Java :

```
open class Animal(val nom: String, var mois: Int) {
    ...

    companion object {
        var nombreNaissances: Int = 0

        fun naissance(nom: String): Animal {
            nombreNaissances++
            return Animal(nom, 0)   // constructeur
        }
    }
}
```

Ajoutez ceci à la fin, pour tester : 

```
val monchiot = Animal.naissance("tifilou")
println("Je possède maintenant ${monchiot}.")
println("J'ai vu ${Animal.nombreNaissances} naissances.")
```

8. Travail à rendre (obligatoire)

Avec le navigateur de fichiers, ouvrez `~/AndroidStudioProjects/TP2` (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`, celui qui est en rouge dans la figure 7 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 .