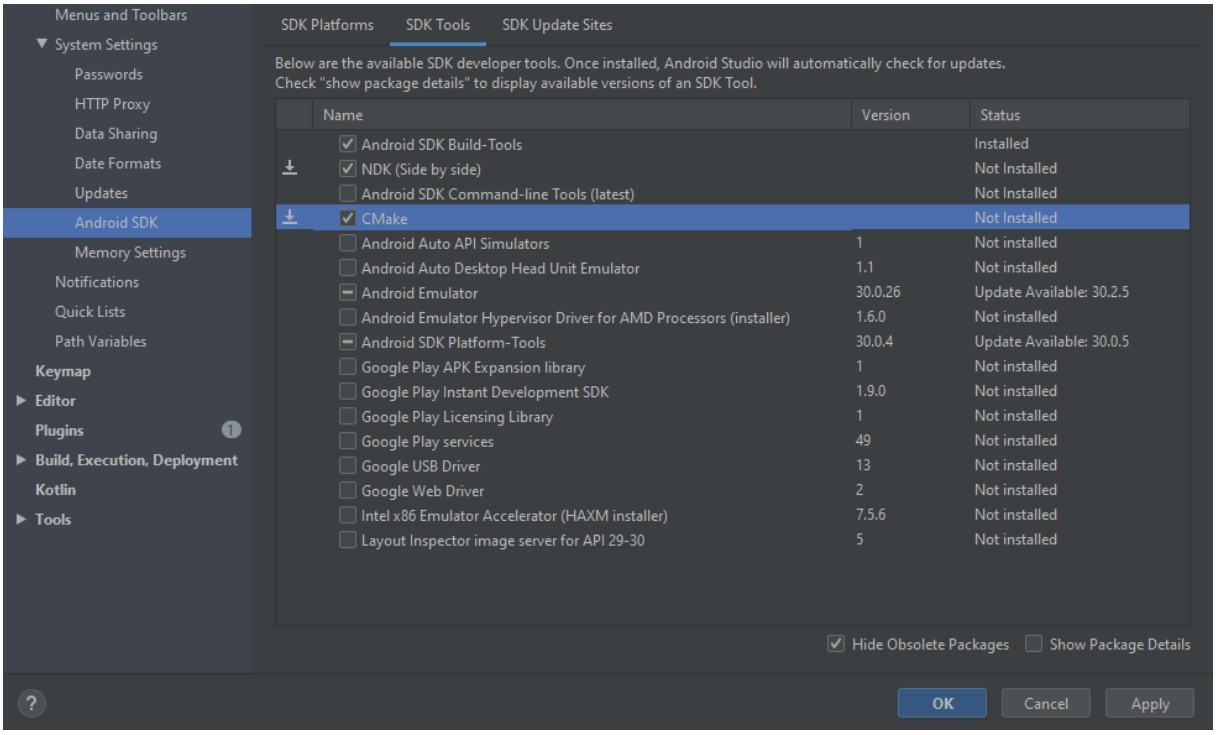
TP4 – Utilisation de JNI

1) Installation du NDK Android

Il nous faut installer le NDK Android afin de pouvoir utiliser la JNI. Pour installer les composants nécessaires, il faut depuis Android Studio aller dans **Tools > SDK Manager** pour ouvrir la fenêtre de configuration du SDK. Nous allons ensuite dans l'onglet **SDK Tools** puis nous sélectionnons les options **CMake** et **NDK (Side by side)**.



En validant, le téléchargement et l'installation de CMake et du NDK est lancée.

2) Création d'une application de type NDK

Lors de la création d'un nouveau projet Android Studio, à la place de choisir « Basic Activity » ou « Empty activity », nous choisissons le template de projet « Native C++ ». Nous laissons le standard C++ par défaut.

En place d'une activité principale, le projet créé possède une première librairie native C++ et un CMakeLists.txt qui va permettre de décrire comment compiler la librairie.

3) Création de l'interface graphique

Pour créer l'interface graphique, nous avons utilisé les outils proposés par Android Studio pour la conception graphique.

L'interface se compose principalement de plusieurs *LinearLayout* imbriqués. Nous avons un layout vertical principal qui va contenir tous les éléments de l'interface et les emboîter, de manière verticale.

Les boutons *READ* et *WRITE* ainsi que le champ numérique qui suit sont alignés à l'aide d'un *LinearLayout* horizontal.

Pour afficher les quatre boutons *UP*, *LEFT*, *RIGHT* et *DOWN*, nous avons utilisé trois *LinearLayout* horizontaux et centrés qui vont afficher les boutons suivant les points cardinaux.

4) Fonctions natives

Le code de la librairie native s'écrira intégralement dans le fichier *native-lib.cpp* situé dans le dossier *cpp* des sources du projet.

La forme des fonctions natives est un peu différente de ce qu'on peut voir d'habitude dans le c++. Dans le cadre de la fonction d'exemple générée automatiquement avec le fichier, nous voyons qu'il faut nommer les fonctions avec le chemin complet du packages, de l'activité et du nom de la fonction, séparés par des underscores.

L'autre spécificité de ces fonctions natives est l'utilisation des primitives JNI pour faire correspondre les types C et les paramètres qui seront passés en Java.

Pour la méthode nativeRead qui prend par exemple un entier en paramètre, nous nous retrouverons avec la signature suivante :

```
Java_com_sidawylepy_myapplication_MainActivity_nativeRead(
        JNIEnv* env,
jobject /* this */, jint number)
```

Avec le premier paramètre JNIEnv* qui correspond à l'environnement JNI (utilisé pour l'interaction avec la machine virtuelle et la manipulation d'objets Java), un second paramètre *jobject* qui correspond lui au « *this* » du C++. Enfin, nous retrouvons le nombre que nous souhaiterons passer en paramètre depuis Java, de type *jint* qui correspond à l'équivalent d'un entier long C.

Le corps de la méthode est plus classique dans le cadre de notre traitement.

```
int resNumber = number * number;
std::string str = "READ : ";
str = str + std::to_string(resNumber);

return  env->NewStringUTF(str.c_str());
```

Nous concaténons le nombre mis à la puissance 2 passé en paramètre à la chaîne de sortie que l'on souhaite, puis nous créons grâce à l'environnement JNI passé en paramètre une chaîne de caractère avec la chaîne C++.

Concernant la méthode pour traduire le texte d'un bouton en japonais, une seule méthode native a été construite prenant le nom du bouton en paramètre (sous forme de *jstring*). On transforme cette chaîne de caractère en *const char** C++ à l'aide de la méthode *GetStringUTFChars* donné par l'objet de l'environnement *env*. A partir de là, on peut réaliser des comparaisons avec la méthode *strcmp* et renvoyer le texte correspondant en japonais.

```
extern "C" JNIEXPORT jstring JNICALL
Java_com_sidawylepy_myapplication_MainActivity_nativeJapaneseButton(
        JNIEnv* env,
        jobject /* this */, jstring buttonName) {

    const char* nativeButtonName = env->GetStringUTFChars(buttonName, 0);

    jstring res = env->NewStringUTF("-");

    if(strcmp(nativeButtonName,"Left") == 0)
    {
        res = env->NewStringUTF("左");
    }
    else if(strcmp(nativeButtonName,"Right") == 0)
    {
        res = env->NewStringUTF("右");
    }
    else if(strcmp(nativeButtonName,"Up") == 0)
    {
        res = env->NewStringUTF("上");
    }
    else if(strcmp(nativeButtonName,"Down") == 0)
    {
        res = env->NewStringUTF("下");
    }

    return res;
}
```

5) Appel des fonctions native depuis l'activité

Une fois ces fonctions natives écrites, il nous reste à les appeler depuis les clics de boutons de l'activité.

Dans un premier temps, nous indiquons à l'activité l'existence de fonctions natives disponibles. Pour cela, pour chaque fonction, on ajoute sa signature dans le corps de la classe de l'activité de cette manière :

```
public native String nativeRead(int number) ;
```

De cette manière, l'appel à cette fonction appellera automatiquement la fonction native du même nom située dans le fichier C++.

Ensuite, nous avons créé plusieurs méthodes qui sont appelés au clic sur un bouton correspondant. Pour les méthodes *READ* and *WRITE*, nous récupérons la valeur entrée par l'utilisateur puis nous appelons la fonction native.

```
public void btnClickRead(View view) {
    int number = getUserNumber();
    TextView tv = findViewById(R.id.sample_text);
    tv.setText(nativeRead(number));
}
```

Le système est similaire pour le bouton *WRITE* : seule la fonction native appelée est différente.

Pour les clics sur les boutons de direction, nous n'avons faite qu'une seule méthode. Le texte du bouton cliqué est récupéré à l'aide de la vue passée en paramètre, puis nous appelons la fonction native qui retourne le texte en japonais avec le nom du bouton passé en paramètre.

```
public void btnClickDirection(View view) {
    TextView tv = findViewById(R.id.sample_text);
    tv.setText(nativeJapButton (((Button)view).getText().toString()));
}
```