Jean-Yves Tigli, Stéphane Lavirotte, Erick Gallesio, Hélène Collavizza
Frédéric Rallo
Polytech'Nice – Sophia
SI3
2021-2022

TD n°9
Introduction Java et Code Natif : JNI

1 Introduction

En grand nombre d'environnements logiciels fournissent deux niveaux de programmation :

- Un niveau où le code est « managé » grâce à une machine virtuelle et un *Framework* spécifique
- Un niveau où le code est plus proche de la configuration matérielle de la cible et s'exécute directement en binaire cible. Il est alors dit « natif ».

Dans tous les cas, une partie du *Framework* managé est écrit en code natif et le rajout de nouvelles entrées/sorties nécessite l'écriture de bibliothèques natives et leur interfaçage avec du code managé. Par exemple, dans l'environnement .Net Framework c'est le `P/Invoke` qui permet cela. Dans l'environnement `Java SE` c'est JNI (Java Native Interface) que nous allons étudier dans ce TD ou JNA (Java Native Access). JNI ou JNA peuvent vous être aussi très utiles pour développer une application en délégant certains calculs à une bibliothèque déjà écrite en C ou C++ (par exemple, pour l'arithmétique de précision ou les calculs en nombres flottants).

Il est donc possible de coupler du Java à du code C ou C++, au sein d'une même application. Le programme démarrera toujours via une machine virtuelle Java, mais certaines méthodes de votre programme pourront être codées via le langage C/C++.

Il vous faut bien entendu disposer de Java dans votre environnement de travail. Si vous êtes dans la machine virtuelle, vous pouvez facilement installer la version suivante :

```
sudo apt update
sudo apt install openjdk-11-jdk
java --version
```

Remarque : Vous serez amenés dans les questions de ce TD à exécuter des programmes (ex. `javac`), à inclure des fichiers (ex. `jni.h`, `jni_md.h`), à déployer des bibliothèques (ex. `System.Loadlibrary`) dont les emplacements ne sont pas forcément a priori déclarés dans les variables d'environnement `PATH`, `INCLUDE_PATH`, `LD_LIBRARY_PATH`.

Sachez que la commande `locate` vous permettra de trouver l'emplacement d'un répertoire ou d'un fichier avec son nom (vous pouvez avoir besoin de l'installer avec `sudo apt install locate`, puis faire `sudo updatedb` avant la première utilisation) :

```
locate nom_fichier_ou_dossier
```

Vous pouvez ainsi compléter les variables d'environnement comme dans l'exemple suivant :

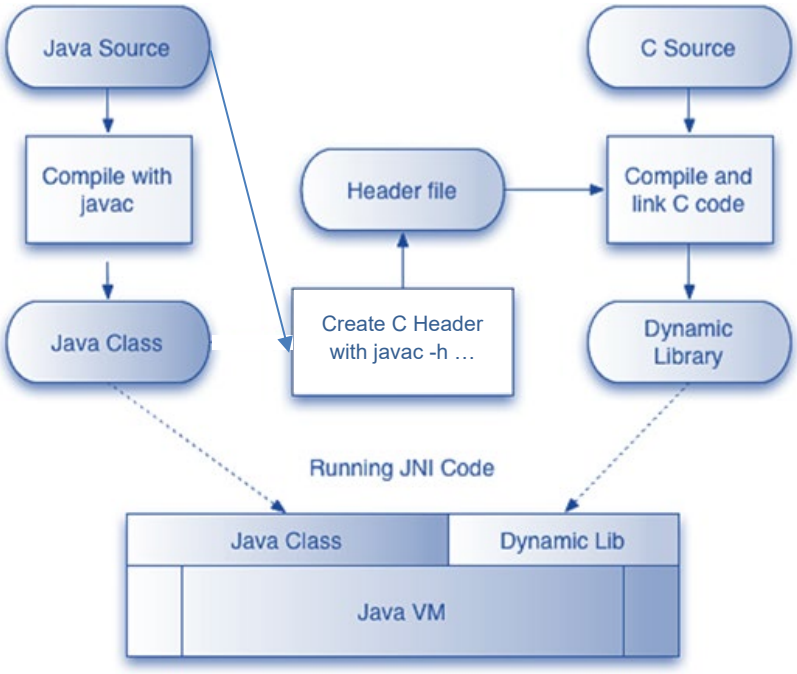
```
export PATH=$PATH:.
```

Jean-Yves Tigli, Stéphane Lavirotte, Erick Gallesio, Hélène Collavizza
Frédéric Rallo
Polytech'Nice – Sophia
SI3
2021-2022

TD n°9
Introduction Java et Code Natif : JNI

2 Couplage simple entre du code Java et du code C++.

Comme bien souvent, nous allons commencer par un exemple très simple : pourquoi pas un petit « Hello World » ? Mais dans notre cas, les choses seront un peu plus subtiles : une partie de l'affichage sera réalisée par du code Java, l'autre par du code C, au travers une bibliothèque dynamique. Pour mener à terme notre exemple, plusieurs tâches sont requises : étudions-les en détail.



2.1 Codage de la partie Java

Pour la partie Java, les choses sont très simples. Sauf que certaines méthodes de la classe ne sont effectivement pas codées en Java : elles sont natives. En Java, une méthode native se code un peu de manière identique à celle d'une méthode abstraite : en effet elle n'a pas de corps et son prototype est terminé par un point-virgule. Par contre il faut préfixer la déclaration de la méthode du mot clé **native**. L'exemple suivant vous propose un exemple de code Java comportant une méthode native.

```
public class HelloWorld {
    public static native void printCpp(); // Déclaration prototype méthode native

    static {
        System.loadLibrary("HelloWorld"); // Chargement de la bibliothèque
    }

    public static void main(String args[]) {
        System.out.print("Hello ");       // Affiche Hello en Java
        HelloWorld.printCpp();            // Affiche World en C/C++
    }
}
```

La première ligne de la classe permet de charger le code pour les méthodes qualifiées de natives. Ce code est stocké soit dans un fichier « `HelloWorld.dll` » (sous Windows) soit dans un fichier « `libHelloWorld.so` » (sous Unix). La méthode native est aussi définie comme étant statique. C'est normal. En effet, le `main` est une méthode statique : comme nous n'avons pas créé d'objet de type `HelloWorld`, le main ne peut alors qu'invoquer qu'une méthode statique. Cela fonctionne aussi avec des méthodes non statiques, bien entendu.

Notez aussi, que pour appeler une méthode native, il n'y a rien d'autre à faire hormis le fait de coder simplement l'appel. A partir de là, les choses restent simples : il vous faut compiler la classe. Si vous n'utilisez pas d'environnement de développement particulier, utilisez le compilateur `javac`, tout comme s'il s'agissait d'un programme Java classique. A ce stade, 50% du programme est écrit et compilé. Il ne nous reste plus qu'à écrire la partie C/C++.

TD n°9
Introduction Java et Code Natif : JNI

2.2 Génération du fichier d'en-tête C/C++

Pour coder la partie C/C++, nous allons dans un premier temps nous intéresser au codage du fichier de déclaration. Deux techniques sont utilisables :

- Soit vous êtes un véritable pro de JNI et dans ce cas, vous pouvez tout coder à la main : mais honnêtement cela n'a pas grand intérêt.
- Soit vous générez automatiquement le fichier en question. Pour ce faire, le JDK vous fournit l'option `-h` à `javac` : `javac -h`. Celui-ci se lance sur le fichier des sources java.

```
javac -h <directory> HelloWorld.java
```

Une fois la commande exécutée, le fichier « `HelloWorld.h` » a dû être généré dans le répertoire *<directory>* que vous avez spécifié en paramètre à l'option `-h`.

Voici un extrait du fichier que vous devriez obtenir par cette génération automatique :

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
* Class: HelloWorld
* Method: printCpp
* Signature: ()V
*/
JNIEXPORT void JNICALL Java_HelloWorld_printCpp(JNIEnv *, jclass);
#ifdef __cplusplus
}
#endif
#endif
```

Vous devez comprendre l'intérêt d'utiliser le générateur de .h étant donné le code produit (même si vous pouvez le faire à la main, mais cela n'a pas d'intérêt). Et encore nous n'avons, pour cette classe, qu'une seule et unique méthode native…

Quelques explications sont ici nécessaires :

- Le premier paramètre `JNIEnv *env` référence toujours l'environnement par lequel vous obtiendrez les fonctions du support JNI.
- Le second paramètre `jobject j` ou `jclass j` donne un complément d'information sur le type de la méthode : si ce paramètre est de type `jclass` nous avons alors à faire à une méthode statique, sinon (type `jobject`) nous avons une méthode normale (il faut un objet pour l'invoquer).
- Les autres paramètres sont en accord avec les paramètres de la méthode Java (voir les exercices suivants).
- Attention : le fichier `jni.h` (du `#include <jni.h>`) inclut lui-même le fichier `jni_md.h` dépendant de l'OS ciblé (`#include "jni_md.h"`). Ce dernier se situe dans le répertoire <OS> juste au-dessus du répertoire contenant `jni.h`. Pour rajouter ce chemin dans la recherche des includes, n'oubliez pas de poser l'option `-I …/<OS>` dans la ligne de compilation.
- on peut voir aussi qu'on utilise le préprocesseur C/C++ pour savoir si le fichier est compilé par un compilateur C ou C++. Vous pouvez admettre ici que le `extern "C"` est nécessaire pour accéder à des fonctions écrites en C, lorsqu'on les compile avec un compilateur C++.

TD n°9
Introduction Java et Code Natif : JNI

2.3 Codage du fichier d'implémentation C/C++

Maintenant, c'est à vous de faire. Il vous faut coder la partie C/C++. Pour ce faire, copiez les prototypes à partir du fichier de déclaration, cela évitera les fautes de frappes. Puis codez le contenu de chaque méthode. Encore une fois, dans ce premier exemple, les choses sont simples : il n'y a qu'une seule fonction à coder et elle réalise simplement une sortie sur la console. N'oubliez pas, bien entendu, d'inclure les fichiers de déclarations suivants : `"HelloWorld.h"` et `<stdio.h>`.

Créez un fichier `HelloWorld.cpp` qui contiendra l'implémentation de l'interface définie dans `HelloWorld.h` (qui a été extrait à partir des méthodes natives de votre classe `HelloWorld`).

```
#include "HelloWorld.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloWorld_printCpp(JNIEnv *env, jclass cls) {
    printf("World !\n");
}
```

Maintenant tout le code nécessaire à l'exécution du programme est fourni. Il ne reste plus qu'à générer la bibliothèque à chargement dynamique attendue par votre classe Java. Regardons cette étape de plus près.

2.4 Génération d'une bibliothèque à chargement dynamique

Pour générer la bibliothèque sous Linux, il faut savoir que le compilateur va devoir trouver différents fichiers de déclarations. Il va donc falloir informer votre environnement de développement que certains fichiers se trouvent dans des répertoires particuliers. Au niveau du JDK, le support JNI se trouve localisé dans le répertoire d'installation du JDK et dans le sous-dossier `include`. Cette manipulation est bien-entendu spécifique à chaque environnement de développement (différente sous Linux, Windows et MacOS)

Voici quelques rappels de ce que nous avons vu dans le TD sur les bibliothèques partagées.

2.4.1 Sous environnements Microsoft

Sous ces environnements, une bibliothèque à chargement dynamique (Dynamique Linking Library) porte l'extension de fichier « `.dll` ». Pour notre projet, le fichier devra donc s'appeler « `HelloWorld.dll` ». Vous devrez pour cela créer un projet « Visual C++ / Win32 / Dynamic Linked Library » (empty project) dans lequel vous insérerez vos fichiers sources (**spécifiez bien un fichier avec l'extension `.cpp` pour que Visual Studio utilise le compilateur C++ et pas C**).

Si vous utilisez Visual Studio, vous pouvez indiquer des répertoires additionnels pour la localisation des fichiers de déclarations (attention, ce n'est pas parce que vous ajoutez des .h à un projet que celui est bien trouvé dans les chemins des `include` pour le compilateur). Pour ce faire, il vous suffit de vous rendre dans les propriétés du projet et d'ajouter les répertoires souhaites aux « Répertoires Include » (dans la rubrique Répertoires VC++ des propriétés de Configuration). Dans mon cas, j'ajoute les répertoires suivants :

```
C:\Program Files\Java\jdk-11.0.7\include; C:\Program Files\Java\jdk-11.0.7\include\win32;
$(IncludePath)
```

2.4.2 Sous environnements Unix

Sous ces environnements, le nom d'une bibliothèque à chargement dynamique commence par les trois lettres « `lib` » et se termine par l'extension « `.so` ». Pour notre projet, le nom de fichier sera donc « `libHelloWorld.so` ».

TD n°9
Introduction Java et Code Natif : JNI

Si vous utilisez le compilateur `g++`, vous pouvez par le biais d'un paramètre sur la ligne de commande, indiquer les répertoires additionnels pour aller chercher les fichiers à `include`. Utilisez pour ce faire, l'option `-I`. N'oubliez pas qu'il faut aussi indiquer que l'on produit une bibliothèque à chargement dynamique en ajoutant l'option `-shared`.

```
g++ -c -fpic -Wall HelloWorld.cpp -I Reps_de_jni -I Reps_de_jni/(win32|linux)
g++ -Wl,-soname,libHelloWorld.so -shared -o libHelloWorld.so HelloWorld.o
```

2.4.3 Sous environnement MacOS

Sous MacOS, le nom d'une bibliothèque à chargement dynamique commence par les trois lettres « `lib` » et se termine par l'extension « `.so` ». Pour notre projet, le nom de fichier sera donc « `libHelloWorld.so` ».

Pour générer la bibliothèque, il suffit de lancer la commande suivante :

```
g++          -c          -fpic          -Wall          HelloWorld.cpp          -
I/Library/Java/JavaVirtualMachines/jdk1.8.0_131.jdk/Contents/Home/include       -
I/Library/.../darwin
g++ -dynamiclib -o libHelloWorld.dylib HelloWorld.o
```

2.5 Exécution du programme

Pour pouvoir exécuter votre programme, il suffit que l'endroit d'où vous lancez la machine virtuelle Java soit celui où la bibliothèque à chargement dynamique et le fichier « `HelloWorld.class` » se trouvent. Saisissez alors simplement : `java HelloWorld`. L'application doit démarrer et les affichages doivent se faire sur la console.

Attention sous Linux au positionnement de votre variable d'environnement `LD_LIBRARY_PATH`, elle doit contenir le répertoire dans lequel se situe votre bibliothèque, souvent le répertoire courant :

```
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
export LD_LIBRARY_PATH
```

Exercice n°1:

Lancer votre programme Java sous Linux en ayant votre bibliothèque dynamique associée (Vous ferez pour l'environnement Windows un peu plus tard).

2.6 Chargement et déchargement de la bibliothèque dynamique

Exercice n°2:

Modifiez votre programme Java afin qu'il attende la saisie d'une chaîne de caractère (cela nous permettra d'en contrôler la terminaison).

3 Echanger des données entre le code Java et le code C/C++

Bien entendu, ces exemples restent très simples. Dans de nombreux cas, vous aurez besoin de faire communiquer vos deux parties de codes de manière plus subtile. Le JDK vous fournit un support impressionnant qui a pour but de vous simplifier la tâche dans ces cas de figures. Etudions tout cela de plus près.

Même si le langage Java et le langage C/C++ se ressemblent, ils sont différents sur bien des points, et notamment sur les types de données. En effet, les noms de types de données ou la syntaxe utilisée pour manipuler certains d'entre eux est très similaire, voir quasi identique. Mais une chose sépare fondamentalement les deux langages : la taille des types de données.

En effet, en C (et donc en C++) seul `sizeof(char)` est garanti (ça vaut `1` octet). Pour tous les autres types, cela dépend du processeur sur lequel va tourner votre programme. Cela a été défini ainsi pour des aspects de rapidité

d'exécution. Ainsi sur processeur Intel core i7, `sizeof(int)` et `sizeof(long)` sont identiques et renvoient la valeur `4` (`4` octets). Pour un autre type de processeur, ces valeurs changeront peut-être.

En Java, il en va autrement. N'oubliez pas que l'intérêt principal de ce langage est la portabilité de l'exécutable généré. Pour ce faire, il faut garantir que la taille des types de données soit la même, quelle que soit la plate-forme sur laquelle le programme tourne. La machine virtuelle Java (JVM) garantit que le type sera soit directement utilisé au niveau du processeur, soit que plusieurs sous-instructions en langage machine seront utilisées pour simuler le type faisant défaut. Ainsi, en Java, un `char` prend deux octets, un `int` prend quatre octets et un `long` en prend huit.

Alors comment faire cohabiter vos codes écris à partir de ces deux langages ?

C'est le JDK qui va vous aider. Si vous retournez plus haut dans ce document, vous verrez que le fichier de déclaration qui a été généré par `javac -h` inclut lui-même un autre fichier de déclaration : le fichier `<jni.h>`. C'est à partir de lui que tout le support JNI est prédéfini.

Prenons pour exemple l'échange de chaînes de caractères en Java et C/C++ et vice versa.

Les difficultés pour les chaînes de caractères, c'est qu'elles peuvent être représentées en ASCII pour le langage C/C++ et en Unicode pour Java. Il faut donc à chaque échange (qu'importe le sens) générer une nouvelle chaîne adaptée au langage récepteur. Grâce à l'environnement JNI, des fonctions de conversions vous sont proposées, comme le montre l'exemple qui suit.

Mais attention à un détail important : en Java, une chaîne de caractères est un objet. Or tout objet est « surveillé » par le garbage collector pour savoir si oui ou non il peut être supprimé de la mémoire. Il doit en être de même pour les chaînes de caractères que vous renvoyez en la JVM. Encore une fois, les fonctions fournies par l'environnement JNI (dont les déclarations se trouvent dans le fichier `jni.h`) règlent le problème.

```
JNIEXPORT void JNICALL
Java_HelloWorld_printStringToCpp(JNIEnv *env, jclass cl, jstring str)
{
    // Construction d'une chaîne C/C++ à partir d'une chaîne Java
    const char *cStr = env->GetStringUTFChars(str,0);
    // Affichage de la chaîne C++
    printf("%s\n", cStr);
    // Libération de la chaîne C++
    env->ReleaseStringUTFChars(str,cStr);
}

JNIEXPORT jstring JNICALL
Java_HelloWorld_stringFromCpp(JNIEnv *env, jobject obj)
{
    // Construction d'une chaîne Java à partir d'une chaîne C/C++
    return env->NewStringUTF("Chaîne en C");
}
```

Exercice n°3:

Ecrivez maintenant un programme Java qui utilise ces deux méthodes. Ajoutez ces deux méthodes à votre bibliothèque C. Vous n'oublierez pas de régénérer le .h à l'aide de `javac -h` pour exporter les fonctions et pour permettre de vérifier la cohérence des prototypes à la compilation.

TD n°9
Introduction Java et Code Natif : JNI

4 Invocation Java depuis le code natif C/C++

4.1 Déterminer les chaînes de description de prototypes

Obtenir la chaîne de description d'un prototype n'est pas très compliqué. Un petit outil vous est fourni dans le JDK. Il s'appelle `javap` (Java Prototype). Cet outil accepte des options pour configurer son résultat. Notamment, l'option `-s` permet d'obtenir l'affichage sous la forme entendue par JNI (et non pas en syntaxe Java). Essayez la commande qui suit. Attention, pour arriver à ses fins, l'outil utilise l'introspection Java : il travaille donc le fichier de bytecode (le fichier compilé). Il ne faut donc pas remettre l'extension ".class" à la suite du nom du fichier.

```
javap -s -p HelloWorld
```

De plus, n'oubliez pas que le fichier de déclaration généré par `javac -h` contient lui aussi chacun des prototypes des méthodes que vous codez en C/C++.

Le tableau suivant rappelle les éléments syntaxiques utilisés pour exprimer un prototype de méthode.

Codage	Signature
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	Double
LclassName	indique une classe - ex: Ljava/lang/String; pour la classe String
[type	indique un tableau - ex: [I pour un tableau d'entier
(args-types)retType	indique un prototype de méthode - ex : (Ljava/lang/String)V

4.2 Invoquer une méthode Java depuis C/C++

L'exemple de code suivant vous montre comment, à partir d'un code C/C++, invoquer une méthode sur un objet Java. La principale difficulté réside dans le fait que le langage Java accepte la surcharge. Le seul nom de la méthode ne suffit donc pas à identifier la méthode à invoquer. Il faut aussi préciser son prototype complet. Pour ce faire, une syntaxe particulière vous est proposée.

Une fois la méthode identifiée, l'environnement JNI vous retournera un identificateur unique de méthode. C'est lui qui servira réellement lors de l'appel.

```
JNIEXPORT void JNICALL
Java_HelloWorld_callJavaMethod(JNIEnv *env, jobject obj) {
    // Récupération d'un objet de Metaclasse
    jclass cls = env->GetObjectClass(obj);
    // Calcule de l'identificateur de "void test(String str)"
    jmethodID mid = env->GetStaticMethodID(cls,"test","(Ljava/lang/String;)V");

    if (mid == 0) {
        // Ca a planté !!!
        fprintf(stderr, "Ouille, ça a planté !");
```

```
    } else {
        // Tout va bien, l'appel peut aboutir.
        jstring str = env->NewStringUTF("Ceci est un paramètre créé en C/C++");
        env->CallVoidMethod(obj,mid,str);
    }
    return;
}
```

Exercice n°4:

Ecrivez maintenant un programme Java qui utilise la méthode `callJavaMethod` telle qu'elle est définie ici. En lisant le code fourni, vous verrez que celle-ci appelle la méthode statique `test` qui prend un paramètre `String`. Vous devrez donc définir cette méthode pour pouvoir faire fonctionner l'appel.

4.3 Accéder à un attribut de la classe Java

La manière d'accéder, à partir de C/C++, à un attribut d'une classe Java est assez similaire à la façon d'invoquer une méthode. Il vous faut en donner son prototype pour en récupérer son identificateur. Regardez de plus près l'exemple suivant.

```
JNIEXPORT jstring JNICALL
Java_HelloWorld_toString(JNIEnv *env, jobject obj) {
    char buffer[256];

    // Obtention de la Metaclasse de HelloWorld
    jclass cls = env->GetObjectClass(obj);

    // Calcul de l'identificateur de l'attribut entier de type int
    jfieldID fid = env->GetFieldID(cls,"entier","I");

    // Récupération de la valeur entière de l'attribut
    int a = env->GetIntField(obj,fid);

    // Modification de la valeur entière de l'attribut
    env->SetIntField(obj, fid, a + 1);

    // Deuxieme récupération de la valeur entière de l'attribut
    a = env -> GetIntField(obj,fid);

    // Génération d'une chaîne contenant la valeur de l'attribut
    sprintf(buffer,"Hello [a = %d]",a);

    // On retourne un objet Java de chaîne de caractères
    return env->NewStringUTF(buffer);
}
```

Exercice n°5:

Ecrivez maintenant un programme Java qui utilise la méthode `toString` telle qu'elle est définie ici. Elle recherche dans la classe un attribut nommé `entier` de type `int`. Attention, vous noterez que pour l'exemple, la valeur de l'attribut est modifiée par la fonction (ce qui n'est pas très académique, mais c'est pour illustrer cette possibilité dans le même exemple).

Exercice n°6:

Soit la fonction `fib` définie en C comme :

```
static int fib(int n) {
```

```
  if (n < 2)
    return n;
  else
    return fib(n-1) + fib(n-2);
}
```

Permettre à votre programme `HelloWorld` d'appeler cette fonction depuis Java.

5 Conclusion.

Au terme de ce TD, nous avons vu qu'il est effectivement possible de coupler du code C/C++ à du code Java. Pour ce faire, le JDK nous fournit le package JNI. Par le biais de ce package, on génère une bonne partie du code nécessaire, mais les choses se compliquent dès lors qu'il y a un échange de données entre les différents environnements. Pour ce faire JNI nous fournit un ensemble de fonctions de conversions et de manipulations.

Notez que les conversions de types (notamment pour les chaînes et les tableaux) coûtent du temps. Il est donc conseillé de minimiser au maximum ces échanges afin de ne pas trop alourdir l'exécution de votre programme. A titre d'exemple, très souvent JNI est utilisé pour intégrer des appels systèmes utilisables depuis C/C++, non accessibles depuis Java. Nous verrons cela lors du prochain et dernier TD.