

TD Interface JAVA / Code Natif : JNI

1 Introduction

En grand nombre d'environnements logiciels embarqués 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.

Dans l'environnement .Net Framework c'est le P/Invoke qui permet cela. Dans l'environnement J2SE c'est JNI que nous allons étudier dans ce TD.

Il est donc possible de coupler du Java à du code 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++.

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

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

a) 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 {  
    static { System.loadLibrary("HelloWorld"); }  
  
    public static native void printCpp();  
  
    public static void main(String args[]) {  
        System.out.print("Hello ");  
        HelloWorld.printCpp();  
    }  
}
```

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.

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 niveau des choses, 50% du programme est écrit et compilé. Il ne nous reste plus qu'à nous attaquer à la partie C++.

TD Interface JAVA / Code Natif : JNI

• Génération du fichier d'en-tête C++.

Pour coder la partie 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 je n'en vois pas l'intérêt. Soit vous êtes, comme moi très fainéant et vous générez de manière automatique le fichier en question. Pour ce faire, le JDK vous fournit l'outil `javah`. Celui-ci se lance sur le fichier précédemment compilé. Attention, vous ne travaillez pas sur un fichier de code source donc vous ne rappelez pas l'extension du fichier sur la ligne de commande.

```
>javah HelloWorld
```

Une fois la commande exécutée, le fichier "HelloWorld.h" a dû être généré sur le disque. En voici son contenu : vous comprendrez alors l'intérêt d'utiliser un outil de génération, et encore nous n'avons, pour cette classe, qu'une seule et unique méthode native.

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

#ifdef _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
```

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` 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.

• Codage du fichier d'implémentation C++.

Maintenant, c'est à vous de faire. Il vous faut coder la partie 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 `<iostream.h>`.

TD Interface JAVA / Code Natif : JNI

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

JNIEXPORT void JNICALL Java_HelloWorld_printCpp(JNIEnv *, jclass) {
    // Un affichage sur la console en C++
    cout << "Hello world in C++" << endl;
}
```

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

- **Génération d'une librairie à chargement dynamique.**

Quelle que soit la plate-forme pour laquelle vous allez générer la librairie, 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.

2.1.1.1 Sous environnements Microsoft.

Sous ces environnements, une librairie à chargement dynamique (Dynamic 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 « Win32 Dynamic Linked Library » (empty project) dans lequel vous insérerez vos fichiers sources.

Si vous utilisez Visual Studio 6.0, vous pouvez indiquer des répertoires additionnels pour la localisation des fichiers de déclarations. Pour ce faire, il vous suffit de saisir par le biais du menu « Tools » dans « Options » les chemins d'accès supplémentaires au niveau de la zone "Additional include directories".

2.1.1.2 Sous environnements Unix.

Sous ces environnements, le nom d'une librairie 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".

Si vous utilisez le compilateur g++, vous pouvez par le biais d'un paramètre sur la ligne de commande, indiquer les répertoires additionnels. Utilisez pour ce faire, l'option -I. Il vous faut de plus indiquer que l'on veut générer une librairie à chargement dynamique : utilisez l'option -shared.

```
>g++ -o libHelloWorld.so -shared -IvotreRep -IRepsdejni HelloWorld.cpp
```

- **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 librairie 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 librairie.

Lancer votre programme java sous Windows et sous Unix en ayant dans chacun des cas créé votre librairie dynamique associée.

TD Interface JAVA / Code Natif : JNI

• Chargement et déchargement de la librairie dynamique

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

Ainsi sous windows vous pourrez, après avoir installé le programme PrcView (recherchez-le sur le Web), vérifier que :

- La librairie dynamique se charge au lancement de votre programme java et se décharge à la fin de son exécution
- Qu'un compteur associé à cette librairie dynamique est bien incrémentée lorsque vous lancez plusieurs fois votre programme. Il s'agit donc bien d'une librairie partagée.

3 Echanger des données entre le code Java et le code 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++ 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, ça 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 Pentium, `sizeof(int)` et `sizeof(long)` sont identiques et renvoient la valeur 4 (4 octets). Pour un autre 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 écrits 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 l'outil `javah` inclut lui-même un autre fichier de déclaration : le fichier `<jni.h>`. C'est à partir de lui que tout le support JNI est pré-défini.

Prenons pour exemple l'échange de chaînes de caractères.

Les difficultés pour les chaînes de caractères, c'est qu'elles peuvent être représentées en ASCII pour le langage C++ (`sizeof(char) == 1`) 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 règlent le problème dont les déclarations se trouvent dans le fichier `jni.h`.

```
JNIEXPORT void JNICALL
```

TD Interface JAVA / Code Natif : JNI

```
Java_HelloWorld_test(JNIEnv *env, jclass, jstring str)
{
    // Construction d'une chaîne C++ à partir d'une chaîne Java
    const char *cStr = env->GetStringUTFChars(str,0);
    // Affichage de la chaîne C++
    cout << cStr << endl;
    // Libération de la chaîne C++
    env->ReleaseStringUTFChars(str,cStr);
}

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

Ecrivez maintenant un programme java qui utilise ces deux méthodes.

4 Invoquer une méthode sur un objet Java.

L'exemple de code suivant vous montre comment, à partir d'un code 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, jclass cls, jobject obj) {
    // Récupération d'un objet de Metaclass
    jclass cls = env -> GetObjectClass(obj);
    // Calcule de l'identificateur de "void test(String param)"
    jmethodID mid = env->GetStaticMethodID(cls,"test","(Ljava/Lang/String;) V");

    if (mid == 0) {
        // Ca a planté !!!
        cout << "Ouille" << endl;
    } else {
        // Tout va bien, l'appel peut aboutir.
        jstring str = env->NewStringUTF("truc");
        env->CallVoidMethod(obj,mid,str);
    }
    return;
}
```

Ecrivez maintenant un programme java qui utilise la méthode callJavaMethod telle qu'elle est définie ici.

- Accéder à un attribut de la classe Java.

La manière d'accéder, à partir de 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 Metaclass de HelloWorld
```

TD Interface JAVA / Code Natif : JNI

```
jclass cls = env -> GetObjectClass(obj);
// Calcule de l'identificateur de l'attribut a de type int
jfieldID fid = env -> GetFieldID(cls,"a","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, 6);

// 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,"Toto [a = %d]",a);

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

Ecrivez maintenant un programme java qui utilise la méthode toString telle qu'elle est définie ici.

- **Déterminer les chaînes de description de prototypes.**

En fait, obtenir la chaîne de description d'un prototype n'est pas si compliqué que cela. En effet, 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'inspection 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 l'outil javah contient lui aussi chacun des prototypes des méthodes que vous codez en C++.

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

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)

5 Mise en œuvre

Nous allons ici traiter un exemple d'application de JNI. Il va nous permettre ici d'accéder aux informations système qui ne sont pas accessible directement depuis Java (ce dernier étant « OS indépendant »).

- Choisissez donc quelques appels systèmes des plus pertinents sous windows et leurs équivalents sous POSIX (exemple : GetProcessId ..etc.).
- Fabriquez deux librairies .so et dll avec la même interface
- Concevez un programme java utilisant alternativement ces deux librairies selon l'Os sur lequel vous trouvez, pour afficher les informations système correspondantes.

TD Interface JAVA / Code Natif : JNI

6 Conclusion.

Au terme de ce TD, nous avons vu qu'il est effectivement possible de coupler du code 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++, non accessibles depuis java.