

TIC – Filière Informatique

Jigé Pont

Nicolas Fuchs

Programmation avancée Java

TP02 – Java Native Interface (JNI)

7/11/2017



Haute école d'ingénierie et d'architecture Fribourg
Hochschule für Technik und Architektur Freiburg

Hes·so

Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Configuration

Système d'exploitation : Windows 10 Famille

Distribution compilateur TDM-GCC sur MinGW 64 bits

1. Un premier exemple

1.1 Cycle de Développement

P1

- 1) Le programme java est déjà écrit. Il est composé de deux classes. La première classe se nomme AclassWithNativeMethods. Elle contient la déclaration de la méthode native (signature) :

```
public native void theNativeMethod();
```

et une méthode publique :

```
public void aJavaMethod() { theNativeMethod(); }
```

qui appelle la méthode native. La deuxième classe se nomme SimpleJNI. Elle contient un bloc statique qui charge la librairie :

```
static {  
    System.loadLibrary("NativeMethodImpl");  
}
```

et la fonction main du programme :

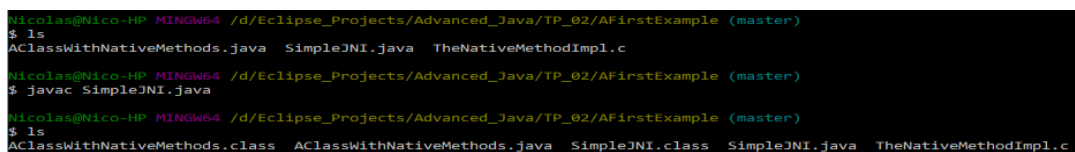
```
public static void main(String[] args) {  
    AclassWithNativeMethods theClass = new AclassWithNativeMethods();  
    theClass.aJavaMethod();    // a NON native method  
}
```

qui instancie la classe AclassWithNativeMethods et fait appel à sa méthode publique aJavaMethod qui elle-même appelle la méthode native.

- 2) On a du ajouter à la variable d'environnement PATH le chemin vers javac.exe. Ensuite, nous avons compilé le code java de la manière suivante :

```
javac SimpleJNI.java
```

On obtient après la compilation java deux fichiers avec l'extension '.class', un pour chaque fichier '.java'.



```
Nicolas@Nico-HP MINGW64 /d/Eclipse_Projects/Advanced_Java/TP_02/AFirstExample (master)  
$ ls  
AclassWithNativeMethods.java SimpleJNI.java TheNativeMethodImpl.c  
Nicolas@Nico-HP MINGW64 /d/Eclipse_Projects/Advanced_Java/TP_02/AFirstExample (master)  
$ javac SimpleJNI.java  
Nicolas@Nico-HP MINGW64 /d/Eclipse_Projects/Advanced_Java/TP_02/AFirstExample (master)  
$ ls  
AclassWithNativeMethods.class AclassWithNativeMethods.java SimpleJNI.class SimpleJNI.java TheNativeMethodImpl.c
```

Visualisation des commandes faites sous l'émulateur bash de git

- 3) On génère le fichier header du code c en appelant la commande suivante :

```
javah -jni AclassWithNativeMethods
```

Il faut spécifier l'option 'jni' pour obtenir le bon type d'exportation :

```
JNIEXPORT void JNICALL Java_AclassWithNativeMethods_theNativeMethod(JNIEnv *, jobject);
```

4) L'implémentation du code C a été fournie par le professeur dans le fichier :
TheNativeMethodImpl.c

5) On a compilé le code C en librairie dll avec la commande suivante :
gcc4jni TheNativeMethodImpl.c NativeMethodImpl.dll
qui génère le fichier NativeMethodImpl.dll

6) Le programme est exécuté.

```
D:\Eclipse_Projects\Advanced_Java\TP_02\AFirstExample>java SimpleJNI  
Hi folks, welcome to the secret world of JNI !
```

P2

Le pointeur env de type JNIEnv* pointe sur une structure qui contient l'interface vers la Java Virtual Machine (JVM). Cette interface contient toutes les fonctions (JNI) nécessaires pour interagir avec la JVM et travailler avec les objets java. Par exemple, si depuis le code java, nous désirons obtenir un texte généré par un code natif (en C), dans ce code natif nous utiliserons la fonction JNI NewStringUTF(env, buffer), buffer étant de type char* pour renvoyer une chaîne de caractères format java.

Attention! Ce pointeur n'est valide que dans son thread associé, le thread duquel la méthode native a été appelée.

P3

Object est une référence sur l'objet java qui a fait l'appel à la fonction native. Ce concept correspond au mot clé this dans le code java.

P4

Les types primitifs java sont définis dans le fichier jni.h qui se trouve à :
C:\Program Files\Java\jdk1.8.0_112\include.

Java type	Native type	Size in bits
boolean	jboolean	unsigned 8 bit
byte	jbyte	signed 8 bit
char	jchar	unsigned 16 bit
short	jshort	signed 16 bit
int	jint	signed 32 bit
long	jlong	signed 64 bit
float	jfloat	32 bit
double	jdouble	64 bit
void	void	-

Src : Programmation avancée JAVA slides part 2 - JNI

P5

Nous avons modifié le code de la classe SimpleJNI.java comme suit :

```
public class SimpleJNI {
    static {
        System.out.println("Line 01");
        System.loadLibrary("NativeMethodImplzzz");
        System.out.println("Line 02");
    }
    public static void main(String[] args) {
        try {
            System.out.println("Line 1");
            AClassWithNativeMethods theClass = new AClassWithNativeMethods();
            System.out.println("Line 2");
            theClass.aJavaMethod(); // a NON native method
            System.out.println("Line 3");
        } catch (Exception e) {
            System.err.println("Test");
            e.printStackTrace();
        }
    }
}
```

Après compilation (javac) et exécution, nous obtenons la sortie suivante sur la console :

```
$ java SimpleJNI
Line 01
Exception in thread "main" java.lang.UnsatisfiedLinkError: no NativeMethodImplzzz in java.library.path
    at java.lang.ClassLoader.loadLibrary(Unknown Source)
    at java.lang.Runtime.loadLibrary0(Unknown Source)
    at java.lang.System.loadLibrary(Unknown Source)
    at SimpleJNI.<clinit>(SimpleJNI.java:13)
```

Nous constatons donc que l'exception est appelée suite à l'appel de la méthode System.LoadLibrary qui est exécutée lors du chargement de la classe en raison du mot clé static.

Nous supprimons 'zzz' ajouté après le nom de la librairie et modifions la classe AClassWithNativeMethods.java comme suit :

```
public class AClassWithNativeMethods {

    public native void theNativeMethodzzz();

    public void aJavaMethod() {
        theNativeMethodzzz(); // the native method
    }
}
```

Après compilation (javac) et exécution, nous obtenons la sortie suivante sur la console :

```
$ java SimpleJNI
Line 01
Line 02
Line 1
Line 2
Exception in thread "main" java.lang.UnsatisfiedLinkError: AClassWithNativeMethods.theNativeMethodzzz()V
    at AClassWithNativeMethods.theNativeMethodzzz(Native Method)
    at AClassWithNativeMethods.aJavaMethod(AClassWithNativeMethods.java:15)
    at SimpleJNI.main(SimpleJNI.java:21)
```

Nous constatons donc que c'est l'appel de la méthode `theClass.aJavaMethod` qui provoque l'exception.

1.2 Passage de paramètres Java

P6

- 1) Modification du code Java :

```
public class AClassWithNativeMethods {  
    public native void theNativeMethod(String str);  
  
    public void aJavaMethod() {  
        theNativeMethod("Test with string parameter"); // the native method  
    }  
}
```

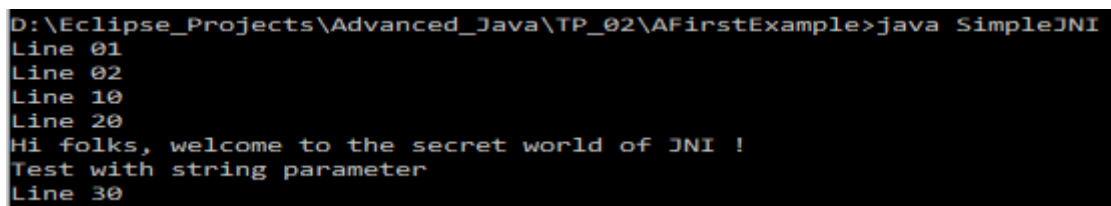
Ajout du paramètre de type String (Java).

- 2) Recompilation du code Java avec la commande `javac`
- 3) Re-génération du header file avec la commande `javah -jni`
- 4) Modification de la méthode native C :

```
JNIEXPORT void JNICALL Java_AClassWithNativeMethods_theNativeMethod(JNIEnv* env, jobject thisObj, jstring str) {  
    const char* c_str = (*env)->GetStringUTFChars(env, str, NULL);  
    printf("Hi folks, welcome to the secret world of JNI !\n");  
    printf("%s\n", c_str);  
    (*env)->ReleaseStringUTFChars(env, str, c_str);  
}
```

Attention! Ne pas oublier la libération mémoire de la chaîne de caractères ☺

- 5) Recompilation de la DLL avec la commande `gcc`
- 6) Lancement du programme java avec la sortie suivante :



```
D:\Eclipse_Projects\Advanced_Java\TP_02\AFirstExample>java SimpleJNI  
Line 01  
Line 02  
Line 10  
Line 20  
Hi folks, welcome to the secret world of JNI !  
Test with string parameter  
Line 30
```

2. Un exemple plus avancé

P7

Résultats :

Le programme wininfo nous renvoie :

- Microsoft Windows 10 (nom du système d'exploitation)
- v1703 (version de mars 2017)
- (64-bit)
- Build 15063 (numéro de compilation)

Le tout est écrit sur une seule ligne.

Au lancement de la commande systeminfo, cette dernière va chercher plusieurs informations, par exemple la première est : chargement des informations du processeur.

Plusieurs autres chargements d'informations sont exécutés par la commande et le résultat final est indiqué dans la figure ci-dessous.

Entre les deux commandes, certaines informations sont communes (nom du système d'exploitation, numéro de compilation). Par contre, le numéro de version v1703 et l'architecture 64-bit du microprocesseur ne sont pas indiqués par la commande systeminfo.

```
D:\Eclipse_Projects\Advanced_Java\TP_02\WinInfo>gcc wininfo.c -o wininfo.exe
D:\Eclipse_Projects\Advanced_Java\TP_02\WinInfo>wininfo
Microsoft Windows 10 v1703 (64-bit), Build 15063

D:\Eclipse_Projects\Advanced_Java\TP_02\WinInfo>systeminfo

Nom de l'hôte: NICO-HP
Nom du système d'exploitation: Microsoft Windows 10 Famille
Version du système: 10.0.15063 N/A version 15063
Fabricant du système d'exploitation: Microsoft Corporation
Configuration du système d'exploitation: Station de travail autonome
Type de version du système d'exploitation: Multiprocessor Free
Propriétaire enregistré: gangsta9s@hotmail.ch
Organisation enregistrée: Hewlett-Packard
Identificateur de produit: 00326-10000-00000-AA774
Date d'installation originale: 14.07.2017, 21:40:41
Heure de démarrage du système: 02.11.2017, 18:11:58
Fabricant du système: Hewlett-Packard
Modèle du système: HP ENVY 17 Notebook PC
Type du système: x64-based PC
Processeur(s): 1 processeur(s) installé(s).
[01] : Intel64 Family 6 Model 69 Stepping 1 GenuineIntel ~2600 MHz
Version du BIOS: Insyde F.34, 19.12.2014
Répertoire Windows: C:\WINDOWS
Répertoire système: C:\WINDOWS\system32
Périphérique d'amorçage: \Device\HarddiskVolume4
Option régionale du système: fr-ch;Français (Suisse)
Paramètres régionaux d'entrée: fr-ch;Français (Suisse)
Fuseau horaire: (UTC+01:00) Amsterdam, Berlin, Berne, Rome, Stockholm, Vienne
Mémoire physique totale: 12 218 Mo
Mémoire physique disponible: 6 649 Mo
Mémoire virtuelle : taille maximale: 14 074 Mo
Mémoire virtuelle : disponible: 7 283 Mo
Mémoire virtuelle : en cours d'utilisation: 6 791 Mo
Emplacements des fichiers d'échange: C:\pagefile.sys
Domaine: WORKGROUP
Serveur d'ouverture de session: \NICO-HP
Correctif(s): 3 Corrections installées.
[01]: KB4022405
[02]: KB4049179
[03]: KB4041676
Carte(s) réseau: 5 carte(s) réseau installée(s).
[01]: VMware Virtual Ethernet Adapter for VMnet1
Nom de la connexion : VMware Network Adapter VMnet1
DHCP activé : Oui
Serveur DHCP : 192.168.226.254
Adresse(s) IP
[01]: 192.168.226.1
[02]: fe80:9574:f62d:703e:d089
[02]: Cisco AnyConnect Secure Mobility Client Virtual Miniport Adapter for Windows x64
Nom de la connexion : Ethernet 2
État : Matériel absent
[03]: VMware Virtual Ethernet Adapter for VMnet8
Nom de la connexion : VMware Network Adapter VMnet8
DHCP activé : Oui
Serveur DHCP : 192.168.74.254
Adresse(s) IP
[01]: 192.168.74.1
[02]: fe80:557f:aa4a:7f7f:d9c5
[04]: Intel(R) Dual Band Wireless-AC 3160
Nom de la connexion : Wi-Fi
DHCP activé : Oui
Serveur DHCP : 1.1.1.1
Adresse(s) IP
[01]: 160.98.127.250
[05]: Realtek PCIe GBE Family Controller
Nom de la connexion : Ethernet
État : Support déconnecté
Configuration requise pour Hyper-V: Extensions de mode du moniteur d'ordinateur virtuel : Oui
Virtualisation activée dans le microprogramme : Oui
Traduction d'adresse de second niveau : Oui
Prévention de l'exécution des données disponible : Oui
```

2.1 Transformation en application JNI

P8

- 1) Aucune modification nécessaire du code fourni par le professeur (ShowWinInfo.java)
- 2) Compilation du code Java avec la commande `javac`
- 3) Génération du header file avec la commande `javah -jni`
- 4) Implémentation du code C (modification du fichier `wininfo.c`) :
 - Ajout d'une ligne : `#include "ShowWinInfo.h"`
 - Ajout de la méthode :

```
JNIEXPORT jstring JNICALL Java_ShowWinInfo_getWinInfo(JNIEnv *env, jclass obj)
{
    // Récupération du TCHAR de la fonction GetOSDisplayString
    TCHAR szOS[BUFSIZE];

    if( !GetOSDisplayString( szOS ) ) // fail
        StringCchCopy(szOS, BUFSIZE, TEXT("NO INFORMATION FOUNDED\n"));

    return (*env)->NewStringUTF(env, szOS);
}
```

- 5) Compilation de la DLL avec la commande `gcc`
- 6) Lancement du programme java avec la sortie suivante :

```
D:\Eclipse_Projects\Advanced_Java\TP_02\WinInfo\jni_v1>java ShowWinInfo
Operating System:
Microsoft Windows 10 v1703 (64-bit), Build 15063
```

2.2 Utilisation de WinInfo pour la valeur de retour

P9

On a du modifier le fichier `ShowWinInfo.java` de la manière suivante :

```
// YOU MAY ONLY MODIFY THE FOLLOWING SINGLE LINE ONLY !
public static native WinInfo getWinInfo();
```

Modification de la valeur de retour de la méthode native `getWinInfo`

On a du créer une nouvelle méthode dans le fichier wininfo.c

```
JNIEXPORT jobject JNICALL Java_ShowWinInfo_getWinInfo(JNIEnv *env, jclass callingObject) {

    jclass winInfoClass = (*env)->FindClass(env, "WinInfo");
    if (!winInfoClass) return (*env)->NewStringUTF(env, "ECHEC");

    jmethodID midConstructor = (*env)->GetMethodID(env, winInfoClass, "<init>", "(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V");
    if (!midConstructor) return (*env)->NewStringUTF(env, "ECHEC");

    TCHAR szOS[BUFSIZE];

    jstring type;
    jint build = -1;
    jbyte arch = -1;

    if( !GetOSDisplayString( szOS ) ) // fail
    {
        StringCchCopy(wininfo_type, BUFSIZE, TEXT("NO INFORMATION FOUNDED\n"));
        type = (*env)->NewStringUTF(env, wininfo_type);
    } else
    {
        type = (*env)->NewStringUTF(env, wininfo_type);
        // Un peu sauvage mais fonctionnel :D
        build = atoi(wininfo_build+6);
        wininfo_arch[3] = '\n';
        arch = atoi(wininfo_arch+1);
    }

    jstring edition = (*env)->NewStringUTF(env, wininfo_edition);
    jstring sp = (*env)->NewStringUTF(env, wininfo_sp);
    jobject winInfoObject = (*env)->NewObject(env, winInfoClass, midConstructor, type, edition, sp);
    if (!winInfoObject) return (*env)->NewStringUTF(env, "ECHEC");

    jfieldID fid = (*env)->GetFieldID(env, winInfoClass, "build", "I");
    if (!fid) return (*env)->NewStringUTF(env, "ECHEC");

    (*env)->SetIntField(env, winInfoObject, fid, build);
    fid = (*env)->GetFieldID(env, winInfoClass, "arch", "B");
    if (!fid) return (*env)->NewStringUTF(env, "ECHEC");

    (*env)->SetByteField(env, winInfoObject, fid, arch);

    return winInfoObject;
}
```

Dans le code ci-dessus, nous avons du utiliser beaucoup de fonctions JNI :

FindClass(env, "WinInfo"), qui nous donne la structure de la classe WinInfo

GetMethodID(env, winInfoClass, "<init>", "(Ljava/lang/String;Ljava/lang/String;Ljava/lang/String;)V"), qui nous donne un identifiant correspondant au constructeur de la classe.

NewObject(env, winInfoClass, midConstructor, type, edition, sp), qui instancie la classe winInfo.

GetFieldID(env, winInfoClass, "build", "I"), qui nous un identifiant correspondant à l'attribut de la classe.

SetIntField(env, winInfoObject, fid, build), qui permet d'affecter une valeur à l'attribut de l'objet.

A chaque fois qu'un retour NULL empêcherait le bon déroulement de la suite du code, nous avons testé ce cas et, en cas de valeur NULL, nous renvoyons directement un jstring avec un message d'erreur.

La méthode getWinInfo peut très bien retourner un String Java qui est un objet.

Nous avons extrait les cinq variables suivantes de la méthode originale GetOSDisplayString pour les passer en visibilité globale.


```
// Cheat pour obtenir les valeurs séparées à la sortie de la fonction GetOSDisplayString
static TCHAR wininfo_type[BUFSIZE] = {0};
static TCHAR wininfo_edition[BUFSIZE] = {0};
static TCHAR wininfo_sp[40] = {0};
static TCHAR wininfo_build[40] = {0};
static TCHAR wininfo_arch[10] = {0};
```

Ainsi nous gardions un accès sur ces variables suite à l'appel de la fonction.

P10

Dans la partie jni_v1, la méthode native retourne un jstring. Dans la partie jni_v2, dans le main, la méthode getWinInfo est censée retourner un objet de type WinInfo.

L'objet String hérite de Object qui implémente la méthode toString. Par conséquent, par polymorphisme, la méthode toString sera appelée correctement.