***TIC – Filière Informatique***

**Jigé Pont**

**Nicolas Fuchs**

**Programmation avancée Java**

**TP02 – Java Native Interface (JNI)**



7/11/2017

# 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

## 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

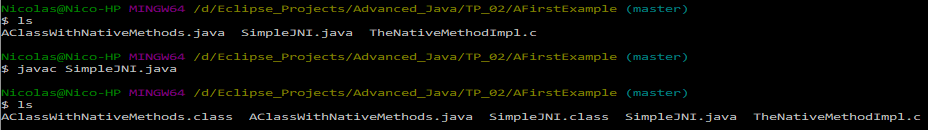
}

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

1. 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'.



Visualisation des commandes faites sous l'émulateur bash de git

1. 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);

1. L'implémentation du code C a été fournie par le professeure 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

1. Le programme est exécuté.



## 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.

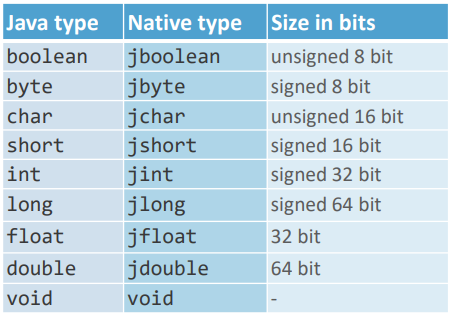
## P3

Jobject 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 :

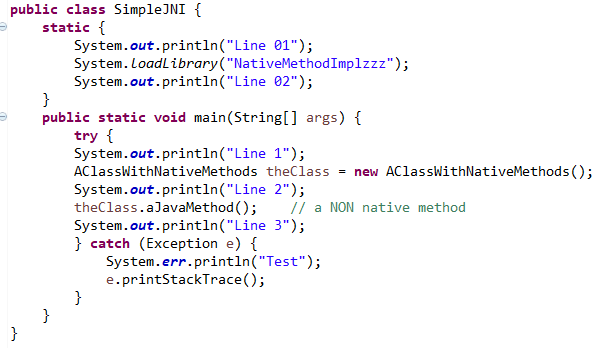
## C:\Program Files\Java\jdk1.8.0\_112\include.



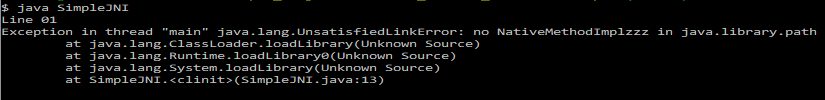
Src : Programmation avancée JAVA slides part 2 - JNI

## P5

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

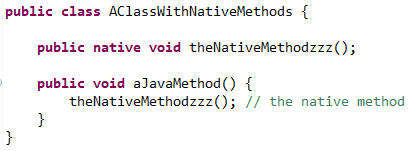


Après compilation (javac) et exécution, nous obtenons la sortie suivante sur la console :

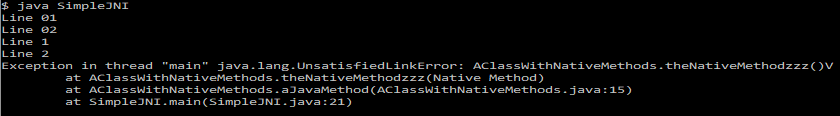


Nous constatons donc que l'exception est appelée suite à l'appel de la méthode System.LoadLibrary qui est exécutée lors du chargement de la classe en raison du mot clé static.

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



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

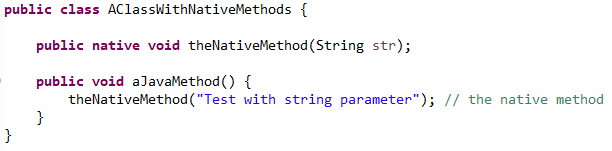


Nous constatons donc que c'est l'appel de la méthode theClass.aJavaMethod qui provoque l'exception.

## Passage de paramètres Java

## P6

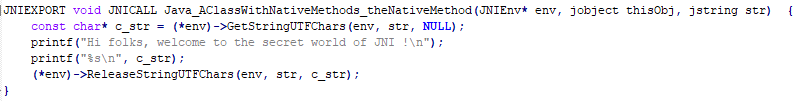
## 1) Modification du code Java :



Ajout du paramètre de type String (Java).

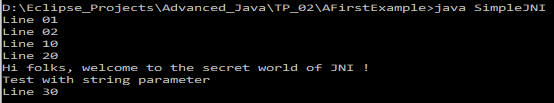
2) Recompilation du code Java avec la commande javac

3) Re-génération du header file avec la commande javah -jni

4) Modification de la méthode native C :

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 :

# 2. Un exemple plus avancé

## P7

Résultats :

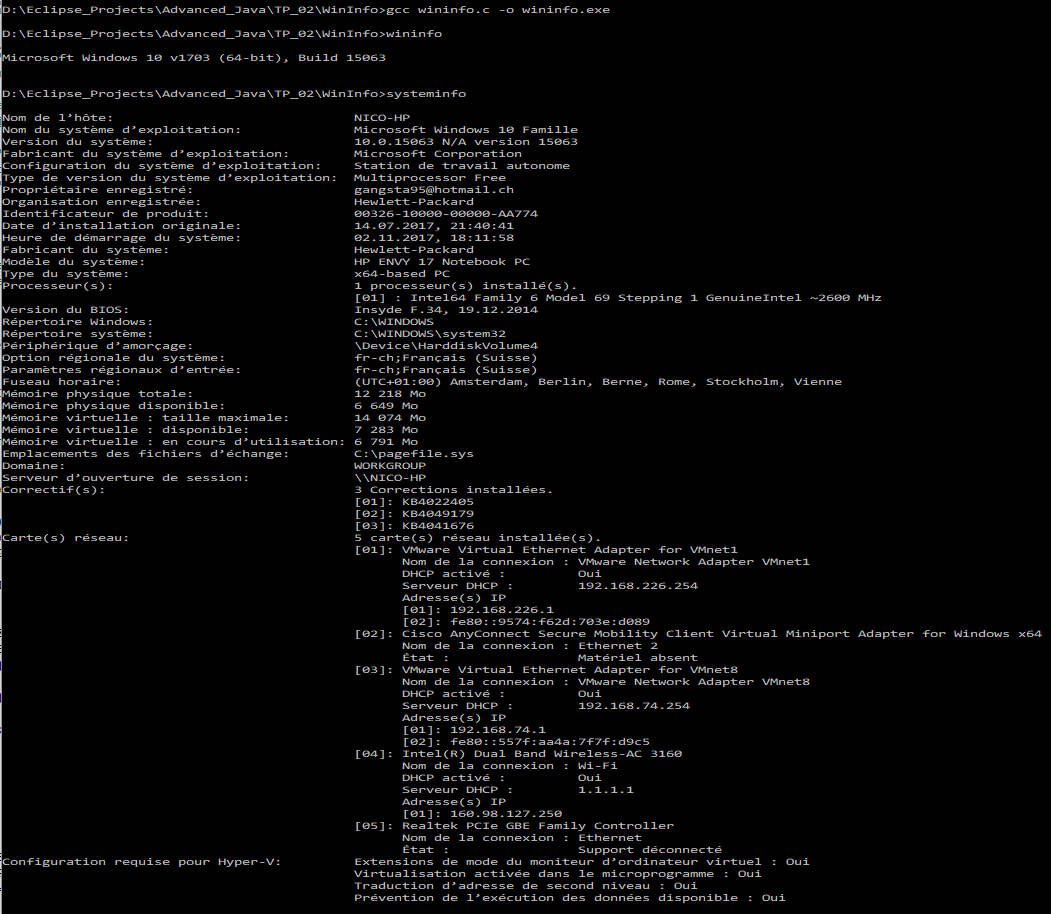
Le programme wininfo nous renvoie :

* 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.

## 

## 2.1 Transformation en application JNI

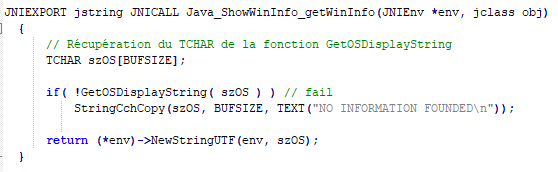
## P8

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 :

5) Compilation de la DLL avec la commande gcc

6) Lancement du programme java avec la sortie suivante :

## 2.2 Utilisation de WinInfo pour la valeur de retour

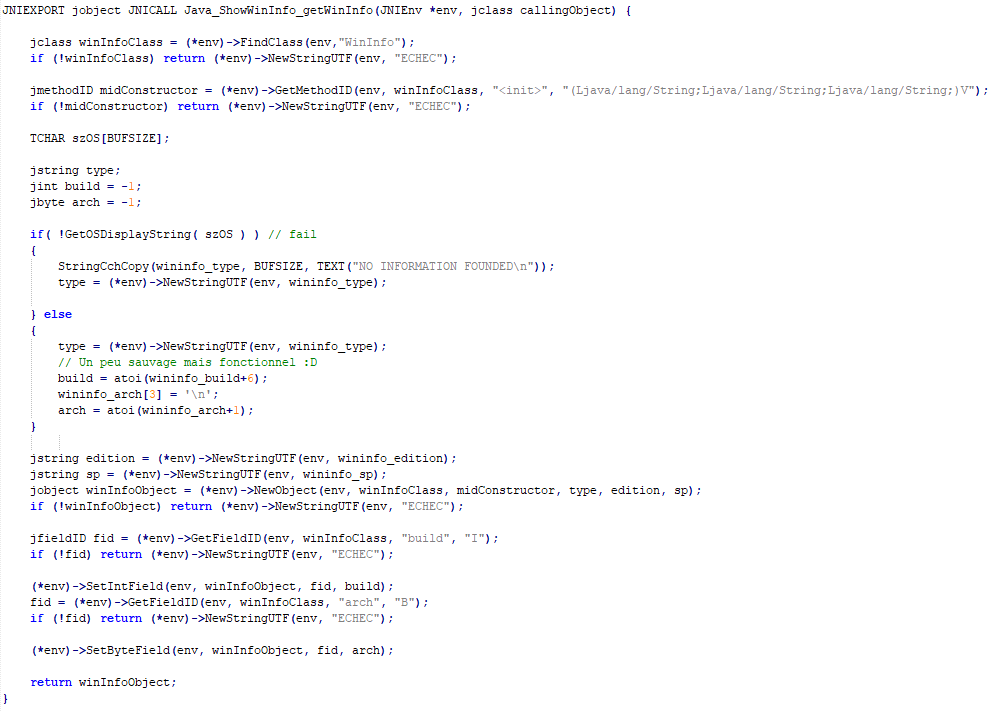
## P9

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



Modification de la valeur de retour de la méthode native getWinInfo

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



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.

## 

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.