



Ecole d'ingénieurs et d'architectes de Fribourg
Hochschule für Technik und Architektur Freiburg

JNI TP02

TAPR2 - Rapport

Filière informatique

Classe I3

Auteurs :

Jocelyn Duc
jocelyn.duc@edu.hefr.ch

Andrea Marcacci
andrea.marcacci@edu.hefr.ch

Enseignant :

Dr Rudolf Scheurer
rudolf.scheurer@hefr.ch

Fribourg, 18 novembre 2012

Table des matières

P1	Première compilation	2
P2	JNIEnv*	3
P3	Paramètre jobject	3
P4	Types JNI	4
P5	Librairie ou méthode manquante	4
P6	Passage d'un String en paramètre	5
P7	Compilation C pure	7
P8	Retourner un String	7
P9	Retourner un objet	8
P10	jni_v1 et jni_v2	11
P11	[Facultatif] Package osinfo	11
	OSXInfo	12
	Makefile	12

P1 Première compilation

Suivez les 6 étapes pour préparer votre première application JNI (sans documenter). Si vous n'utilisez pas gcc4jni.bat indiquez votre environnement de travail et la commande utilisée pour la compilation de la librairie partagée.

Nous n'avons pas utilisé le fichier gcc4jni.bat. Voici notre configuration :

Mac OSX 10.7.5

java version 1.6.0_35

Java(TM) SE Runtime Environment (build 1.6.0_35-b10-428-11M3811)

Java HotSpot(TM) 64-Bit Server VM (build 20.10-b01-428, mixed mode)

Notre marche à suivre pour la compilation de l'application :

– Compilation JAVA

Premièrement, il faut compiler la partie Java.

Compilation Java

```
1 javac SimpleJNI.java
```

– Génération du fichier d'entête

Ensuite, il faut générer le fichier d'entête. Cette opération est à faire uniquement sur les fichiers qui possèdent des méthodes natives. Dans notre cas, seul le fichier `AClassWithNativeMethods.java` contient une telle méthode.

Fichier d'entête

```
1 javah -jni AClassWithNativeMethods
```

– Compilation du code natif

La compilation du code natif en C se fait par le biais de ces 2 commandes :

Compilation C

```
1 cc -c -lx -I/System/Library/Frameworks/JavaVM.framework/Headers  
  TheNativeMethodImpl.c  
2 cc -dynamiclib -o libNativeMethodImpl.jnilib TheNativeMethodImpl.  
  o -framework JavaVM
```

– Lancement de l'application

Le `java.library.path` est spécifiée à l'exécution pour que la librairie externe soit trouvée.

Exécution

```
1 java -Djava.library.path=. SimpleJNI
```

– Résultat

Le résultat affiché lors de l'exécution est le suivant :

Sortie console

```
1 Hi folks, welcome to the JNI world!
```

P2 JNIEnv*

Quelle est l'utilité du paramètre de type JNIEnv passé à la méthode native ?
A quelle structure pointe ce paramètre ?*

Le paramètre JNIEnv* offre la possibilité d'accéder aux fonctions JNI. Il s'agit de l'interfaçage entre le code natif et la JVM. JNIEnv* est un pointeur qui pointe vers une structure qui pointe elle-même vers une structure qui contient les fonctions qui permettent l'interaction entre le code natif et la JVM. Ces fonctions et autres types utilisés sont définis dans les fichiers jni.h et jni_md.h.

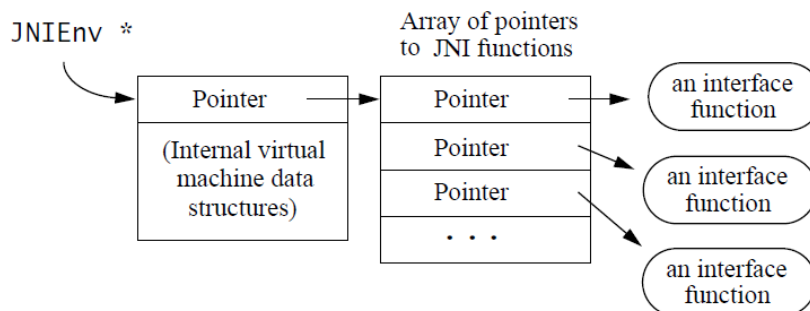


FIGURE 1 – Architecture de JNIEnv

P3 Paramètre jobject

Quelle est l'utilité du paramètre de type jobject passé à la méthode native ?

Le paramètre jobject représente l'objet Java qui a appelé la méthode native. Dans l'exemple suivant, thisObj est une référence vers l'objet Java appelant, de type AClassWithNativeMethod.

TheNativeMethodImpl.c

```

1 JNIEXPORT void JNICALL Java_AClassWithNativeMethods_theNativeMethod
2   (JNIEnv* env, jobject thisObj) {
3     printf("Hi folks, welcome to the JNI world!\n");
4   }

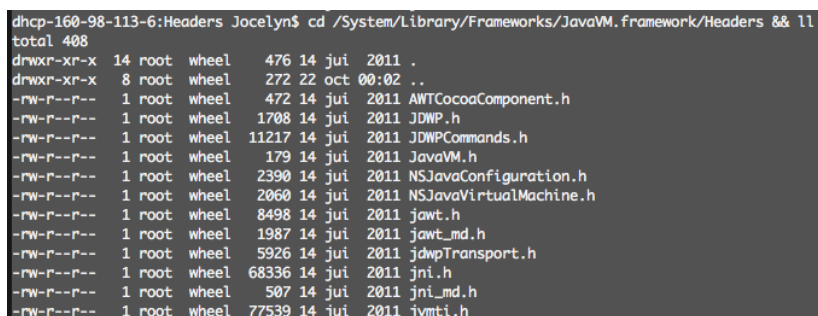
```

P4 Types JNI

Dans quels fichiers sont définis les types comme jstring, jint, etc. ? Où se trouvent ces fichiers ?

Les types JNI tels que jstring, jclass ou jarray (etc.) sont définis dans les fichiers d'entête jni.h et jni_md.h.

Les fichiers jni.h ou jni_md.h se trouvent, sur nos OS, dans /System/Library/Frameworks/JavaVM.framework/Headers



```

dhcp-160-98-113-6:Headers Jocelyn$ cd /System/Library/Frameworks/JavaVM.framework/Headers && ll
total 408
drwxr-xr-x  14 root  wheel   476 14 jui   2011 .
drwxr-xr-x   8 root  wheel   272 22 oct  00:02 ..
-rw-r--r--   1 root  wheel   472 14 jui   2011 AWT CocoaComponent.h
-rw-r--r--   1 root  wheel  1708 14 jui   2011 JDWP.h
-rw-r--r--   1 root  wheel 11217 14 jui   2011 JDWPCommands.h
-rw-r--r--   1 root  wheel   179 14 jui   2011 JavaVM.h
-rw-r--r--   1 root  wheel  2390 14 jui   2011 NSJavaConfiguration.h
-rw-r--r--   1 root  wheel  2060 14 jui   2011 NSJavaVirtualMachine.h
-rw-r--r--   1 root  wheel  8498 14 jui   2011 jawt.h
-rw-r--r--   1 root  wheel  1987 14 jui   2011 jawt_md.h
-rw-r--r--   1 root  wheel  5926 14 jui   2011 jdpTransport.h
-rw-r--r--   1 root  wheel 68336 14 jui   2011 jni.h
-rw-r--r--   1 root  wheel   507 14 jui   2011 jni_md.h
-rw-r--r--   1 root  wheel 77539 14 jui   2011 jvmti.h

```

FIGURE 2 – Contenu du dossier Headers

P5 Librairie ou méthode manquante

Documenter et expliquer les différences entre les exceptions lancées dans les deux situations suivantes :

a

Chargement d'une librairie qui n'existe pas

Dans cette première variante, la nous tentons d'appeler et donc de charger une librairie qui n'existe pas.

SimpleJNI.java

```

1 static {
2   System.loadLibrary("FakeLibrary");
3 }

```

Le `ClassLoader` qui lance une exception de type `UnsatisfiedLinkError`, dans ce cas. On comprend clairement que la librairie demandée ne peut être chargée car non trouvée.

Sortie console

```
1 Exception in thread "main" java.lang.UnsatisfiedLinkError: no
  FakeLibrary in java.library.path
2   at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1872)
3   at java.lang.Runtime.loadLibrary0(Runtime.java:845)
4   at java.lang.System.loadLibrary(System.java:1084)
5   at SimpleJNI.<clinit>(SimpleJNI.java:12)
6 make: *** [run] Error 1
```

b

Appel d'une méthode native qui n'existe pas (chargement de la librairie OK)

Si le chargement de la librairie a lieu mais qu'on tente d'y appeler une méthode qui n'existe pas, une erreur sensiblement différente est levée. La librairie est tout de même chargée sans problème mais la méthode reste non trouvée.

Sortie console

```
1 Exception in thread "main" java.lang.UnsatisfiedLinkError:
  AClassWithNativeMethods.theNativeMethod()V
2   at AClassWithNativeMethods.theNativeMethod(Native Method)
3   at AClassWithNativeMethods.aJavaMethod(AClassWithNativeMethods.
    java:15)
4   at SimpleJNI.main(SimpleJNI.java:16)
5 make: *** [run] Error 1
```

P6 Passage d'un String en paramètre

Adapter l'application afin de permettre le passage d'un String (de Java vers le code C) qui doit être affiché à la place du «Hello World ...». Documenter et expliquer les changements nécessaires.

L'idée est d'appeler une méthode Java qui, via le mécanisme de JNI, appeler une méthode native en C pour afficher du texte depuis l'implémentation en C.

Premièrement, nous déclarons la méthode native et la méthode Java nécessaire pour l'appel :

AClassWithNativeMethods.java

```
1 public class AClassWithNativeMethods {
2
3   public native String sayHello(String message);
4 }
```

```
5 public void saySomething() {
6     System.out.println(sayHello("Hello world!"));
7 }
8 }
```

Nous appelons ensuite cette méthode depuis SimpleJNI :

SimpleJNI.java

```
1 public class SimpleJNI {
2
3     static {
4         System.loadLibrary("NativeMethodImpl");
5     }
6
7     public static void main(String[] args) {
8         AClassWithNativeMethods theClass = new AClassWithNativeMethods
9             ();
10        theClass.saySomething();
11    }
12 }
```

Finalement, du côté C, il faut ajouter la nouvelle méthode native `sayHello()` à la librairie `TheNativeMethodImpl`.

La signature de cette deuxième méthode sera forcément différente de celle vue dans l'exercice précédent puisque cette méthode native va retourner un `jstring`.

TheNativeMethodImpl.c

```
1 JNIEXPORT jstring JNICALL Java_AClassWithNativeMethods_sayHello
2 (JNIEnv* env, jobject thisObj, jstring myParameter){
3     char buf [128];
4     const char *str = (*env)->GetStringUTFChars(env, myParameter,
5         NULL);
6     printf("%s", str);
7     (*env)->ReleaseStringUTFChars(env, myParameter, str);
8     scanf("%s", buf);
9     return (*env)->NewStringUTF(env, buf);
10 }
```

On notera l'utilisation de quelques méthodes offertes par JNI pour la compatibilité des `String` entre Java et C.

- `GetStringUTFChars(JNIEnv *env, jstring string, jboolean *isCopy)`
Retourne un pointeur vers un tableau de caractères UTF-8 qui représentent le `String`. Ce tableau est valide jusqu'à ce qu'il soit libéré par la méthode `ReleaseStringUTFChars()`.
- `ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf)`
Informe la machine virtuelle Java que le code natif n'a plus besoin d'accéder à `utf`.

- `NewStringUTF(JNIEnv *env, const char *bytes)`
Construit un nouvel objet `java.lang.String` à partir d'un tableau de caractères au format UTF-8.

A l'exécution, nous retrouvons donc notre message retourné et affiché comme suit :

Sortie console

```
1 Hello world!
```

P7 Compilation C pure

Compiler le code C avec `gcc wininfo.c -o wininfo.exe` et documenter l'affichage lors de l'exécution de l'application sur votre ordinateur.

Utilisant tous les deux des MacBook Pro équipés de Mac OSX 10.7.5, nous avons adapté l'exercice et les sources pour obtenir des résultats similaires sur Mac OSX.

Voici l'affichage lors de l'exécution de notre version.

Sortie console

```
1 Operating System:
2 Mac OSX: 10.7.5
```

Ce résultat est atteint en compilant et exécutant directement le fichier `osxinfo.c`.

Compilation

```
1 gcc -framework CoreServices -o osxinfo osxinfo.c
```

Notre implémentation spécifique à Mac OSX se trouve en annexe et est documentée au point `OSXInfo`.

P8 Retourner un String

Transformer l'application C en méthode native avec interface `JNI`, retournant un `String` et utilisable par `ShowWinInfo`. Documenter et expliquer les changements.

La méthode native suivante a été ajoutée. Elle permet de retourner un `String` et peut être appelée depuis notre classe `ShowOSXInfo`.

osxinfo.c

```
1 JNIEXPORT jstring JNICALL Java_ShowOSXInfo_getOSXInfo(JNIEnv *env,
    object thisObj)
2 {
```



```
3   getInfo();
4   char ver[1000];
5   snprintf(ver, sizeof ver, "Mac OSX: %d.%d.%d", maj, min, bug);
6   return (*env)->NewStringUTF(env, ver);
7 }
```

Il faut encore inclure le fichier d'entête `ShowOSXInfo.h`, préalablement générée avec `javah`.

`osxinfo.c`

```
1 #include "ShowOSXInfo.h"
```

Nous obtenons donc les mêmes informations à la sortie via Java que via une compilation et exécution en C pur.

Sortie console

```
1 Operating System:
2 Mac OSX: 10.7.5
```

P9 Retourner un objet

Changer le type de la valeur de retour de la méthode native. Elle devra retourner une instance de la classe `WinInfo` encapsulant les informations concernant le système d'exploitation. Par conséquent il faudra aussi adapter la classe `ShowWinInfo`. Documenter et expliquer les changements (notamment les fonctions `JNI` utilisées et le traitement d'erreur des deux côtés, C et Java).

Le but est maintenant de générer un objet regroupant des données sur le système d'information à partir du code natif et de l'afficher sur Java à l'aide de sa méthode `toString()`. Pour ce faire, nous utiliserons la classe `OSXInfo` que nous avons créé (voir le point `OSXInfo`).

Premièrement, l'objet `OSXInfo.java` est créé. Il contient les informations du système. Il s'agit de l'équivalent de `WinInfo.java` pour Windows.

`OSXInfo.java`

```
1 public class OSXInfo {
2     int maj = 0;
3     int min = 0;
4     int bug = 0;
5     String ver = "";
6
7     public OSXInfo(int maj, int min, int bug, String ver) {
8         this.maj = maj;
9         this.min = min;
10        this.bug = bug;
11    }
12 }
```

```
11     this.ver = ver;
12 }
13
14 @Override
15 public String toString() {
16     return "Mac OS X: " + this.maj + "." + this.min + "." + this.
        bug + "\n(Ver:" + ver + ")";
17 }
18 }
```

Deuxièmement, nous allons adapter le code Java. La méthode native sera `getOSXInfo()`. Elle sera appelée de la classe `ShowOSXInfo`. Comme l'appel natif se fait depuis cette dernière, il faut qu'elle charge la librairie nécessaire :

ShowOSXInfo.java

```
1 static {
2     System.loadLibrary("osxinfo");
3 }
```

La méthode avec le mot-clé `native` est ajoutée dans la classe `ShowOSXInfo` et l'appel natif est fait dans la méthode d'entrée `main()`.

ShowOSXInfo.java

```
1 public static native OSXInfo getOSXInfo();
2
3 public static void main(String[] args) {
4     System.out.println("Operating System:\n" + getOSXInfo());
5 }
```

Pour la suite, il faut reprendre l'implémentation de `osxinfo.c` et lui ajouter la méthode en code natif. Cette méthode pourra alors récupérer les informations du système, créer un objet `OSXInfo`, lui donner les valeurs du système et le retourner, avant d'être affiché du côté Java.

osxinfo.c

```
1 JNIEXPORT jobject JNICALL Java_ShowOSXInfo_getOSXInfo(JNIEnv *env,
    jobject thisObj)
2 {
3     getInfo();
4     jstring ver;
5
6     if (maj = 10){
7         switch (min) {
8             case 1:
9                 ver = (*env)->NewStringUTF(env, "Puma");
10                break;
11             case 2:
12                 ver = (*env)->NewStringUTF(env, "Jaguar");
13                break;
14             case 3:
15                 ver = (*env)->NewStringUTF(env, "Panther");
```

```
16         break;
17     case 4:
18         ver = (*env)->NewStringUTF(env, "Tiger");
19         break;
20     case 5:
21         ver = (*env)->NewStringUTF(env, "Leopard");
22         break;
23     case 6:
24         ver = (*env)->NewStringUTF(env, "Snow Leopard");
25         break;
26     case 7:
27         ver = (*env)->NewStringUTF(env, "Lion");
28         break;
29     case 8:
30         ver = (*env)->NewStringUTF(env, "Mountain Lion");
31         break;
32     default:
33         ver = (*env)->NewStringUTF(env, "Unknown");
34     }
35 }
36 else {
37     ver = (*env)->NewStringUTF(env, "Unknown");
38 }
39
40 jclass osxinfo = (*env)->FindClass(env, "OSXInfo");
41 if (osxinfo == NULL) {
42     jclass exception = (*env)->FindClass(env, "java/lang/
43         Exception");
44     (*env)->ThrowNew(env, exception, "Cannot find OSXInfo");
45     return NULL;
46 }
47 jmethodID constructorMethod = (*env)->GetMethodID(env, osxinfo, "
48     <init>", "(IIILjava/lang/String;)V");
49 if (constructorMethod == NULL) {
50     (*env)->DeleteLocalRef(env, osxinfo);
51     jclass exception = (*env)->FindClass(env, "java/lang/
52         Exception");
53     (*env)->ThrowNew(env, exception, "Cannot find constructor");
54     ;
55     return NULL;
56 }
57
58 jobject result = (*env)->NewObject(env, osxinfo,
59     constructorMethod, maj, min, bug, ver);
60
61 return result;
62 }
```

Le résultat après exécution est le suivant :

Sortie console

```
1 Operating System:
2 Mac OS X: 10.7.5
```

```
3 (Ver:Lion)
```

P10 jni_v1 et jni_v2

Copier le fichier wininfo.dll du répertoire jni_v1 (où la méthode native retourne un string) au répertoire jni_v2. Expliquer pourquoi le programme Java s'exécute quand-même sans erreur. Quelles sont les conséquences concernant le passage d'objets entre JNI et Java ?

Dans la variante jni_v1, un `String` est affiché, simplement. Pour la variante jni_v2, un objet `OSXInfo`, implicitement transtypé en `String` est affiché, grâce à la méthode `toString()`. Donc dans les deux cas, on se retrouve avec un `String` qui s'affiche. Si l'on déplace la librairie `libosxinfo.jnilib` de `jni_v1` vers `jni_v2`, la sortie sera telle qu'à la première partie de l'exercice (`jni_v1`).

Sortie console

```
1 Operating System:
2 Mac OSX: 10.7.5
```

P11 [Facultatif] Package osinfo

[Facultatif] Documenter et expliquer les changements nécessaires si on placerait le code Java dans un package nommé `osinfo`. Où est-ce qu'il faudrait placer la dll ?

Pour ce faire, il faut modifier le nom de la méthode dans le code natif. Le nom du package doit se retrouver entre le nom Java et le nom de la classe.

Remplacer :

osxinfo.c

```
1 JNIEXPORT jobject JNICALL Java_ShowOSXInfo_getOSXInfo(JNIEnv *env,
    jobject thisObj) {}
```

par :

osxinfo.c

```
1 JNIEXPORT jobject JNICALL Java_osinfo_ShowOSXInfo_getOSXInfo(JNIEnv
    *env, jobject thisObj) {}
```

De plus, il faudrait générer les fichiers d'entête en incluant le nom du package :

Génération de fichier d'entête

```
1 javah -jni osinfo.ShowOSXInfo
```

OSXInfo

[Facultatif] Proposer une implémentation équivalente à WinInfo pour Mac OSX

Comme proposé par l'enseignant et puisque nous travaillons tous les deux sur Mac OSX, nous avons décidé de proposer notre version `OSXInfo`, équivalente à `WinInfo`. Le but académique étant d'une part, de faire passer un `jstring` et d'autre part, un `jobject` contenant plusieurs attributs, nous avons préparé une librairie similaire pour Mac OSX.

Version de base de `osxinfo.c`

```
1 #include <CoreServices/CoreServices.h>
2 #include <string.h>
3
4 SInt32 maj, min, bug;
5
6 int getInfo() {
7     Gestalt(gestaltSystemVersionMajor, &maj);
8     Gestalt(gestaltSystemVersionMinor, &min);
9     Gestalt(gestaltSystemVersionBugFix, &bug);
10 }
11
12 int main()
13 {
14     getInfo();
15     printf("Mac Version: %d.%d.%d\n", maj, min, bug);
16 }
```

Les sources complètes, notamment avec les méthodes natives, se trouvent en annexe de ce rapport.

Makefile

[Facultatif] Création d'un fichier Makefile

Afin de gagner du temps lors des différentes compilations après nos différents et fréquents changements dans le code, nous avons décidé de faire des fichiers `Makefile`. Nous vous proposons également les implémentations de ces derniers.

Exemple de Makefile utilisable pour `AFirstExample` et `ASecondExample`

```
1 all:
2     javac SimpleJNI.java
3     javah -jni AClassWithNativeMethods
4     cc -c -I/System/Library/Frameworks/JavaVM.framework/Headers
5         TheNativeMethodImpl.c
6     cc -dynamiclib -o libNativeMethodImpl.jnilib TheNativeMethodImpl.o
7     o -framework JavaVM
8
9 run:
```

```
8  @java -Djava.library.path=. SimpleJNI
9
10 clean:
11  @rm -f *.o
12  @echo Done
13
14 clean_all:
15  @rm -f *.o
16  @rm -f *.h
17  @rm -f *.class
18  @rm -f *.jnilib
19  @echo Done
```