

28.04.01 - version 1.1 :

- Nettoyage du code html (titres, paragraphes, tableaux, images, ancres) par Armel.
- Enlevé de nombreuses balises de type {font}.
- Laissé les {strong} et les {em}.

25.06.2000 - Version 1.0 :

- Dernière mise à jour de la version française

B: L'Interface Native Java [*Java Native Interface*] (JNI)

Le contenu de cette annexe a été fourni et est utilisé avec la permission d'Andrea Provaglio (www.AndreaProvaglio.com).

Le langage Java et son API standard sont suffisamment riches pour écrire des applications complètes. Mais dans certains cas on doit appeler du code non-Java ; par exemple, pour accéder à des fonctionnalités spécifiques du système d'exploitation, s'interfacer avec des matériels particuliers, réutiliser une base de code non-Java existante, ou implémenter des parties de codes à contraintes temps réel fortes.

S'interfacer avec du code non-Java nécessite un support dédié dans le compilateur et dans la machine virtuelle, ainsi que des outils supplémentaires pour associer le code Java au code non-Java. La solution standard fournie par JavaSoft pour appeler du code non-Java est appelée la *Java Native Interface*, qui sera introduite dans cette annexe. Ceci n'est pas un traitement en profondeur, et dans certains cas vous êtes supposés avoir une connaissance partielle des concepts et techniques concernés.

JNI est une interface de programmation assez riche qui permet d'appeler des méthodes natives depuis une application Java. Elle a été ajoutée dans Java 1.1, en maintenant un certain degré de compatibilité avec son équivalent en Java 1.0 : la native method interface (NMI). NMI a des caractéristiques de conception qui la rendent impropre à l'utilisation sur certaines machines virtuelles. Pour cette raison, les versions ultérieures du langage pourraient ne plus supporter NMI, et elle ne sera pas couverte ici.

Actuellement, JNI est conçue pour s'interfacer uniquement avec les méthodes natives écrites en C ou C++. En utilisant JNI, vos méthodes natives peuvent :

- créer, inspecter et modifier des objets Java (y compris les tableaux et les **Strings**),
- appeler des méthodes Java,
- intercepter [*catch*] et générer [*throw*] des exceptions,
- charger des classes et obtenir des informations sur les classes,
- effectuer des contrôles lors de l'exécution (*run-time type checking*).

De ce fait, pratiquement tout ce qu'on peut faire avec des classes et des objets en Java ordinaire, on peut aussi le faire dans des méthodes natives.

Appeler une méthode native

Nous allons commencer avec un exemple simple : un programme Java qui appelle une méthode native, qui à son tour appelle la fonction **printf()** de la bibliothèque C standard.

La première opération consiste à écrire le code Java qui déclare une méthode native et ses arguments :

```

//: appendixb:ShowMessage.java

public class ShowMessage {

    privatenative void ShowMessage(String msg);

    static {

System.loadLibrary("MsgImpl");

        // Astuce Linux, si vous ne pouvez pas
        // positionner le chemin des bibliothèques
        // dans votre environnement :
        // System.load(
        //"/home/bruce/tij2/appendixb/UseObjImpl.so");
    }

    public static void main(String[] args) {

ShowMessage app = new ShowMessage();

app.ShowMessage("Generated with JNI");

    }

} ///:~

```

La déclaration de la méthode native est suivie par un bloc **static** qui appelle **System.loadLibrary()** (qu'on pourrait appeler à n'importe quel moment, mais ce style est plus clair). **System.loadLibrary()** charge une DLL en mémoire et se lie à elle. La DLL doit être dans votre chemin d'accès aux bibliothèques système (*system library path*). L'extension de nom de fichier est automatiquement ajoutée par la JVM dépendant de la plateforme.

Dans le code ci-dessus on peut voir également un appel à la méthode **System.load()**, qui est mis en commentaire. Le chemin spécifié ici est un chemin absolu, plutôt que dépendant d'une variable d'environnement. Utiliser une variable d'environnement est naturellement une meilleure solution et est plus portable, mais si vous n'y arrivez pas vous pouvez mettre en commentaire l'appel à **loadLibrary()** et enlever la mise en commentaire de l'appel à la méthode **System.load()**, en ajustant le chemin de votre propre répertoire.

Le générateur d'entête [*header file generator*] : **javah**

Maintenant, compilez votre fichier source Java et lancez **javah** sur le fichier **.class** résultant, en précisant le paramètre **-jni** (ceci est déjà fait dans le makefile inclus dans la fourniture du code source pour ce livre) :

```
javah -jni ShowMessage
```

javah lit la classe Java et pour chaque déclaration de méthode native il génère un prototype de fonction dans un fichier d'entête C ou C++. Voici le résultat : le fichier source **ShowMessage.h** (légèrement modifié pour l'inclure dans ce livre) :

```
/* DO NOT EDIT THIS FILE
- it is machine generated */

#include <jni.h>

/* Header for class ShowMessage */

#ifndef _Included_ShowMessage
#define _Included_ShowMessage
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:      ShowMessage
 * Method:     ShowMessage
 * Signature:  (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL
Java_ShowMessage_ShowMessage
    (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

Comme on peut le voir par la directive préprocesseur **#ifdef __cplusplus**, ce fichier peut être compilé aussi bien par un compilateur C que C++. La première directive **#include** inclut **jni.h**, un fichier d'entête qui, entre autres choses, définit les types qu'on peut voir utilisés dans le reste du fichier. **JNIEXPORT** et **JNICALL** sont des macros qui une fois étendues génèrent les directives spécifiques aux différentes plateformes. **JNIEnv**, **jobject** et **jstring** sont des définitions de types de données, qui seront expliquées par la suite.

Les conventions de nommage [*name mangling*] et les signatures de fonctions

JNI impose une convention de nommage (appelée *name mangling*) aux méthodes natives. Ceci est important car

cela fait partie du mécanisme par lequel la machine virtuelle lie les appels Java aux méthodes natives. Fondamentalement, toutes les méthodes natives commencent par le mot "Java", suivi par le nom de la méthode native. Le caractère sous-tiret est utilisé comme séparateur. Si la méthode java native est surchargée, alors la signature de fonction est ajoutée au nom également ; on peut voir la signature native dans les commentaires précédant le prototype. Pour plus d'informations sur les conventions de nommage, se référer à la documentation de JNI.

Implémenter votre DLL

Arrivé ici, il ne reste plus qu'à écrire un fichier source C ou C++ qui inclut le fichier d'entête généré par **javah** et implémenter la méthode native, puis le compiler et générer une bibliothèque dynamique. Cette opération dépend de la plateforme. Le code ci-dessous est compilé et lié dans un fichier appelé **MsgImpl.dll** pour Windows ou **MsgImpl.so** pour Unix/Linux (le makefile fourni avec les listings du code contient les commandes pour faire ceci ; il est disponible sur le CD ROM fourni avec ce livre, ou en téléchargement libre sur www.BruceEckel.com) :

```

//: appendixb:MsgImpl.cpp

//# Testé avec VC++ & BC++. Le chemin d'include
//# doit être adapté pour trouver les en-têtes
//# JNI. Voir le makefile de ce chapitre
//# (dans le code source téléchargeable)
//# pour exemple.

#include <jni.h>
#include <stdio.h>
#include "ShowMessage.h"

extern "C" JNIEXPORT> void JNICALL
Java_ShowMessage_ShowMessage(JNIEnv* env,
jobject, jstring jMsg) {
    const char* msg=env->GetStringUTFChars(jMsg,0);
    printf("Thinking in Java, JNI: %s\n", msg);
    env->ReleaseStringUTFChars(jMsg, msg);
    ///:~

```

Les arguments qui sont passés à la méthode native permettent de revenir dans Java. Le premier, de type **JNIEnv**, contient tous les points d'ancrage *[hooks]* (nous allons revenir là-dessus dans la section suivante). Le deuxième argument a une signification différente selon le type de la méthode. Pour des méthodes non-**static** comme dans l'exemple ci-dessus, le deuxième argument est l'équivalent du pointeur "this" en C++ et similaire au **this** en Java : c'est une référence à l'objet qui a appelé la méthode native. Pour les méthodes **static**, c'est une référence à l'objet **Class** où la méthode est implémentée.

Les arguments restants représentent les objets java passés dans l'appel de la méthode native. Les scalaires sont

aussi passés de cette façon, mais en les passant par valeur.

Dans les sections suivantes nous allons expliquer ce code en regardant de quelle façon on accède à la JVM et comment on la contrôle depuis l'intérieur d'une méthode native.

Accéder à des fonctions JNI : l'argument **JNIEnv**

Les programmeurs utilisent les fonctions JNI pour interagir avec la JVM depuis une méthode native. Comme on l'a vu dans l'exemple ci-dessus, chaque méthode native JNI reçoit un argument spécial comme premier paramètre : l'argument **JNIEnv**, qui est un pointeur vers une structure de données JNI spéciale de type **JNIEnv_**. Un élément de la structure de données JNI est un pointeur vers un tableau généré par la JVM. Chaque élément de ce tableau est un pointeur vers une fonction JNI. Les fonctions JNI peuvent être appelée depuis la méthode native en déréférençant ces pointeurs (c'est plus simple qu'il n'y paraît). Chaque JVM fournit sa propre implémentation des fonctions JNI, mais leurs adresses seront toujours à des décalages prédéfinis.

A l'aide de l'argument **JNIEnv**, le programmeur a accès à un large éventail de fonctions. Ces fonctions peuvent être regroupées selon les catégories suivantes :

- obtenir une information de version,
- effectuer des opérations sur les classes et les objets,
- gérer des références globales et locales à des objets Java,
- accéder à des champs d'instances et à des champs statiques [*instance fields and static fields*],
- appeler des méthodes d'instances et des méthodes statiques,
- effectuer des opérations sur des chaînes et des tableaux,
- générer et gérer des exceptions Java.

Il existe de nombreuses fonctions JNI qui ne seront pas couvertes ici. A la place, je vais montrer le principe d'utilisation de ces fonctions. Pour des informations plus détaillées, consultez la documentation JNI de votre compilateur.

Si on jette un coup d'oeil au fichier d'entête **jni.h**, on voit qu'à l'intérieur de la directive préprocesseur **#ifdef __cplusplus**, la structure **JNIEnv_** est définie comme une classe quand elle est compilée par un compilateur C++. Cette classe contient un certain nombre de fonctions inline qui permettent d'accéder aux fonctions JNI à l'aide d'une syntaxe simple et familière. Par exemple, la ligne de code C++ de l'exemple précédent :

```
env->ReleaseStringUTFChars(jMsg, msg);
```

pourrait également être appelée en C comme ceci :

```
(*env)->ReleaseStringUTFChars(env, jMsg, msg);
```

On notera que le style C est (naturellement) plus compliqué ; on doit utiliser un double déréférencement du pointeur **env**, et on doit aussi passer ce même pointeur comme premier paramètre de l'appel à la fonction JNI. Les exemples de cette annexe utilisent le style C++.

Accéder à des chaînes Java

Comme exemple d'accès à une fonction JNI, observez le code dans **MsgImpl.cpp**. Ici, l'argument **JNIEnv env** est utilisé pour accéder à une **String** Java. Les **Strings** Java sont au format Unicode, donc si on en reçoit une et

qu'on veut la passer à une fonction non-Unicode (**printf()**, par exemple), il faut d'abord la convertir en caractères ASCII avec la fonction JNI **GetStringUTFChars()**. Cette fonction prend une **String** Java et la convertit en caractères UTF-8 (ceux-ci font 8 bits pour contenir de valeurs ASCII ou 16 bits pour contenir de l'Unicode ; si le contenu de la chaîne d'origine était composée uniquement d'ASCII, la chaîne résultante sera également de l'ASCII).

GetStringUTFChars() est une des fonctions membres de **JNIEnv**. Pour accéder à la fonction JNI, on utilise la syntaxe typique C++ pour appeler une fonction membre à l'aide d'un pointeur. On utilise la forme ci-dessus pour appeler l'ensemble des fonctions JNI.

Passer et utiliser des objets Java

Dans l'exemple précédent nous avons passé une **String** à la méthode native. On peut aussi passer des objets Java de sa propre création à une méthode native. A l'intérieur de sa méthode native, on peut accéder aux champs *[fields]* et aux méthodes de l'objet reçu.

Pour passer des objets, on utilise la syntaxe Java normale quand on déclare la méthode native. Dans l'exemple ci-dessous, **MyJavaClass** a un champ **public** et une méthode **public**. La classe **UseObjects** déclare une méthode native qui prend un objet de la classe **MyJavaClass**. Pour voir si la méthode native manipule son argument, le champ **public** de l'argument est positionné, la méthode native est appelée, et enfin la valeur du champ **public** est imprimée.

```

//: appendixb:UseObjects.java

class MyJavaClass {
    public int aValue;

    public void divByTwo() { aValue /= 2; }
}

public class UseObjects {
    private native void
changeObject(MyJavaClass obj);

    static {
System.loadLibrary("UseObjImpl");

        // Astuce Linux, si vous ne pouvez pas
        // positionner le chemin des bibliothèques
        // dans votre environnement :
        // System.load(
        //"/home/bruce/tij2/appendixb/UseObjImpl.so");
    }

    public static void main(String[] args) {
UseObjects app = new UseObjects();

MyJavaClass anObj = new MyJavaClass();

```

```

    anObj.aValue = 2;

    app.changeObject(anObj);

    System.out.println("Java: " + anObj.aValue);

}

} ///:~

```

Après avoir compilé le code et exécuté **javah**, on peut implémenter la méthode native. Dans l'exemple ci-dessous, après avoir obtenu les identificateurs du champ et de la méthode, on y accède à l'aide de fonctions JNI.

```

/// appendixb:UseObjImpl.cpp
/// // # Testé avec VC++ & BC++. Le chemin d'include
/// // # doit être adapté pour trouver les en-têtes
/// // # JNI. Voir le makefile de ce chapitre
/// // # (dans le code source téléchargeable)
/// // # pour exemple.

#include <jni.h>

extern "C" JNIEXPORT void JNICALL

Java_UseObjects_changeObject(
    JNIEnv* env, jobject, jobject obj) {

    jclass cls = env->GetObjectClass(obj);

    jfieldID fid = env->GetFieldID(
        cls, "aValue", "I");

    jmethodID mid = env->GetMethodID(
        cls, "divByTwo", "()V");

    int value = env->GetIntField(obj, fid);

    printf("Native: %d\n", value);

    env->SetIntField(obj, fid, 6);

    env->CallVoidMethod(obj, mid);

    value = env->GetIntField(obj, fid);

    printf("Native: %d\n", value);

} ///:~

```

Ignorant l'équivalent de "this", la fonction C++ reçoit un jobject, qui est l'aspect natif de la référence à l'objet Java que nous passons depuis le code Java. Nous lisons simplement **aValue**, l'imprimons, changeons la valeur,

appelons la méthode **divByTwo()** de l'objet, et imprimons la valeur à nouveau.

Pour accéder à un champ ou une méthode Java, on doit d'abord obtenir son identificateur en utilisant **GetFieldID()** pour les champs et **GetMethodID()** pour les méthodes. Ces fonctions prennent la classe, une chaîne contenant le nom de l'élément, et une chaîne donnant le type de l'information : le type de donnée du champ, ou l'information de signature d'une méthode (des détails peuvent être trouvés dans la documentation de JNI). Ces fonctions retournent un identificateur qu'on doit utiliser pour accéder à l'élément. Cette approche pourrait paraître tordue, mais notre méthode n'a aucune connaissance de la disposition interne de l'objet Java. Au lieu de cela, il doit accéder aux champs et méthodes à travers les index renvoyés par la JVM. Ceci permet à différentes JVMs d'implémenter différentes dispositions des objets sans impact sur vos méthodes natives.

Si on exécute le programme Java, on verra que l'objet qui est passé depuis le côté Java est manipulé par notre méthode native. Mais qu'est ce qui est exactement passé ? Un pointeur ou une référence Java ? Et que fait le ramasse-miettes pendant l'appel à des méthodes natives ?

Le ramasse-miettes continue à travailler pendant l'exécution de la méthode native, mais il est garanti que vos objets ne seront pas réclamés par le ramasse-miettes durant l'appel à une méthode native. Pour assurer ceci, des *références locales* sont créées auparavant, et détruites juste après l'appel à la méthode native. Puisque leur durée de vie englobe l'appel, on sait que les objets seront valables tout au long de l'appel à la méthode native.

Comme ces références sont créées et ensuite détruites à chaque fois que la fonction est appelée, on ne peut pas faire des copies locales dans les méthodes natives, dans des variables **static**. Si on veut une référence qui dure tout le temps des appels de fonctions, on a besoin d'une référence globale. Les références globales ne sont pas créées par la JVM, mais le programmeur peut créer une référence globale à partir d'une locale en appelant des fonctions JNI spécifiques. Lorsqu'on crée une référence globale, on devient responsable de la durée de vie de l'objet référencé. La référence globale (et l'objet qu'il référence) sera en mémoire jusqu'à ce que le programmeur libère explicitement la référence avec la fonction JNI appropriée. C'est similaire à **malloc()** et **free()** en C.

JNI et les exceptions Java

Avec JNI, les exceptions Java peuvent être générées, interceptées, imprimées et retransmises exactement comme dans un programme Java. Mais c'est au programmeur d'appeler des fonctions JNI dédiées pour traiter les exceptions. Voici les fonctions JNI pour gérer les exceptions :

- **Throw()** Émet un objet exception existant. Utilisé dans les méthodes natives pour réémettre une exception.
- **ThrowNew()** Génère un nouvel objet exception et l'émet.
- **ExceptionOccurred()** Détermine si une exception a été émise et pas encore effacée [*cleared*].
- **ExceptionDescribe()** Imprime une exception et la trace de la pile [*stack trace*].
- **ExceptionClear()** > Efface une exception en cours.
- **FatalError()** Lève [*raises*] une erreur fatale. Ne revient pas [*does not return*].

Parmi celles-ci, on ne peut pas ignorer **ExceptionOccurred()** et **ExceptionClear()**. La plupart des fonctions JNI peuvent générer des exceptions, et comme le langage ne dispose pas de construction équivalente à un bloc try Java, il faut appeler **ExceptionOccurred()** après chaque appel à une fonction JNI pour voir si une exception a été émise. Si on détecte une exception, on peut choisir de la traiter (et peut-être la réémettre). Il faut cependant s'assurer qu'une exception est effacée en final. Ceci peut être fait dans notre fonction en utilisant **ExceptionClear()** ou dans une autre fonction si l'exception est réémise, mais cela doit être fait.

Il faut s'assurer que l'exception est effacée, sinon les résultats seront imprévisibles si on appelle une fonction JNI alors qu'une exception est en cours. Il y a peu de fonctions JNI qu'on peut appeler sans risque durant une

exception ; parmi celles-ci, évidemment, il y a toutes les fonctions de traitement des exceptions.

JNI et le threading

Puisque Java est un langage multithread, plusieurs threads peuvent appeler une méthode native en même temps (la méthode native peut être suspendue au milieu de son traitement lorsqu'un second thread l'appelle). C'est au programmeur de garantir que l'appel natif est "thread-safe" ; c'est à dire qu'il ne modifie pas des données partagées de façon non contrôlée. Fondamentalement, il y a deux possibilités : déclarer la méthode native **synchronized**, ou implémenter une autre stratégie à l'intérieur de la méthode native pour assurer une manipulation correcte des données concurrentes.

De plus, on ne devrait jamais passer le pointeur JNIEnv à travers des threads, car la structure interne sur laquelle il pointe est allouée thread par thread et contient des informations qui n'ont de sens que dans ce thread particulier.

Utiliser une base de code préexistantes

La façon la plus simple d'implémenter des méthodes JNI natives est de commencer à écrire des prototypes de méthodes natives dans une classe Java, de compiler cette classe, et exécuter le fichier **.class** avec **javah**. Mais si on possède une large base de code préexistante qu'on désire appeler depuis Java ? Renommer toutes les fonctions de notre DLL pour les faire correspondre aux conventions du JNI name mangling n'est pas une solution viable. La meilleure approche est d'écrire une DLL d'encapsulation "à l'extérieur" de votre base de code d'origine. Le code Java appelle des fonctions de cette nouvelle DLL, qui à son tour appelle les fonctions de votre DLL d'origine. Cette solution n'est pas uniquement un contournement ; dans la plupart des cas on doit le faire de toutes façons parce qu'on doit appeler des fonctions JNI sur les références des objets avant de pouvoir les utiliser.

Information complémentaire

Vous pouvez trouver des éléments d'introduction, y compris un exemple C (plutôt que C++) et une discussion des problèmes Microsoft, dans l'Annexe A de la première édition de ce livre, que vos pouvez trouver sur le CD ROM fourni avec ce livre, ou en téléchargement libre sur www.BruceEckel.com. Des informations plus complètes sont disponibles sur java.sun.com (dans le moteur de recherche, sélectionnez "training & tutorials" avec "native methods" comme mots-clés). Le chapitre 11 de *Core Java 2, Volume II*, par Horstmann & Cornell (Prentice-Hall, 2000) donne une excellente description des méthodes natives.