



UNIVERSITÉ
CÔTE D'AZUR

Java Native Interface

Présentation: Stéphane Lavirotte

Auteurs: ... et al*

(*) Cours réalisé grâce aux documents de :
Jean-Michel Douin, Jean-Yves Tigli, Stéphane Lavirotte

Mail: Stephane.Lavirotte@univ-cotedazur.fr

Web: <http://stephane.lavirotte.com/>

Université Côte d'Azur



Machine Virtuelle (d'exécution)

Et p-code ...



Principe du p-code (un vieux concept)

- ✓ Une machine à code p ou une machine à code portable est une machine virtuelle conçue pour exécuter du p-code (un langage assembleur d'un CPU virtuel).
- ✓ Le terme de p-code apparaît pour la première fois au début des années 1970
- ✓ Principale Motivation
 - Construire une machine à pile pour l'exécution
 - Lutter contre la fuite mémoire (alors première source de BUG sur du code « classique »)
- ✓ Pas que Java et CLR mais aussi Matlab, etc.

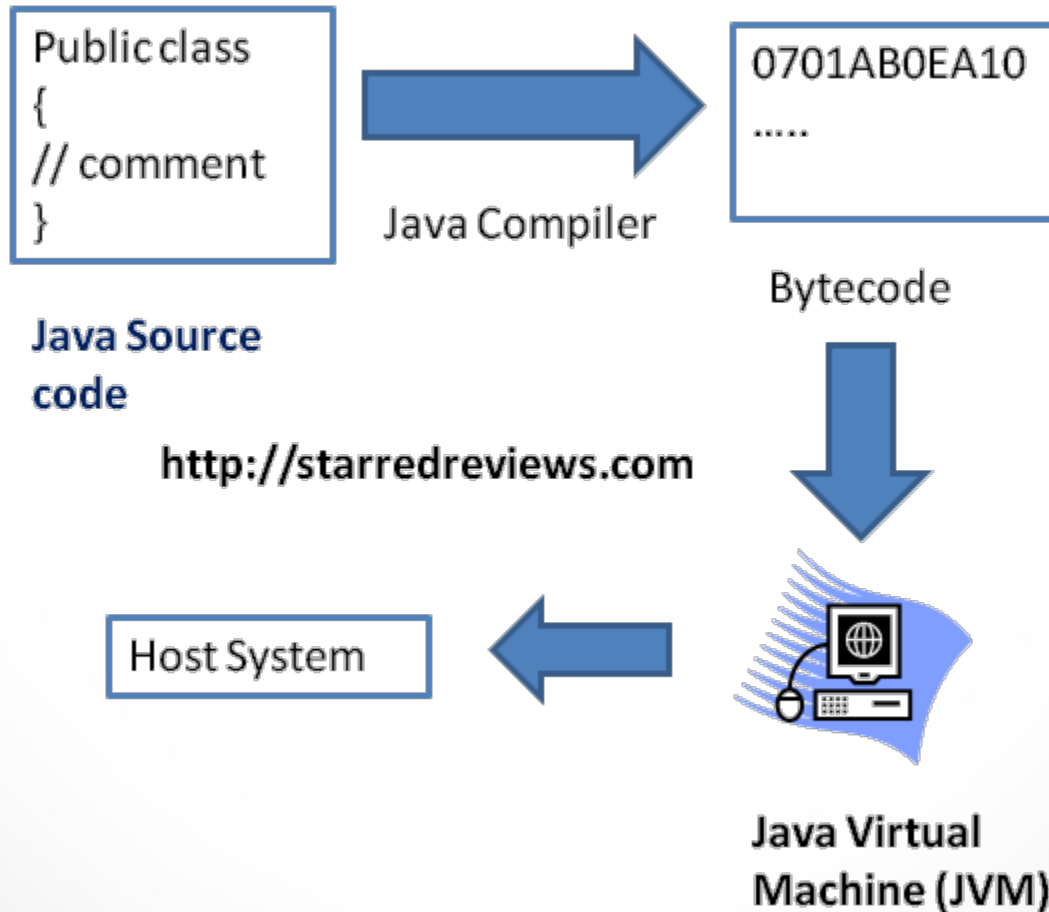


Machine Virtuelle Java

Et bytecode ...



Bytecode en Java pour la JVM





Bytecode en Java pour la JVM

- ✓ **Langage portable** : un programme, une fois compilé, fonctionnera aussi bien sous des stations Unix que sous Windows ou autre, sans aucune modification.
- ✓ **Le code source Java est compilé non pas pour un processeur donné, mais pour une machine virtuelle (c'est-à-dire qui n'a pas d'existence physique), la JVM (Java Virtual Machine).**
- ✓ **Le code résultant est nommé ByteCode.**

Bytecode en Java pour la JVM

- ✓ Lors de l'exécution le ByteCode est transformé en un code machine compréhensible par le processeur de la machine réelle.
- ✓ Java est donc aussi un langage interprété.
- ✓ L'interprète de la JVM est très élaboré pour être le plus rapide possible : il inclut un JIT (Just In Time Compiler) de façon à faire la traduction du Bytecode vers du code natif seulement lorsque c'est nécessaire (première instantiation d'une classe, boucles...).



Machine Virtuelle « .Net » CLR

Et CIL (historiquement MSIL ...)

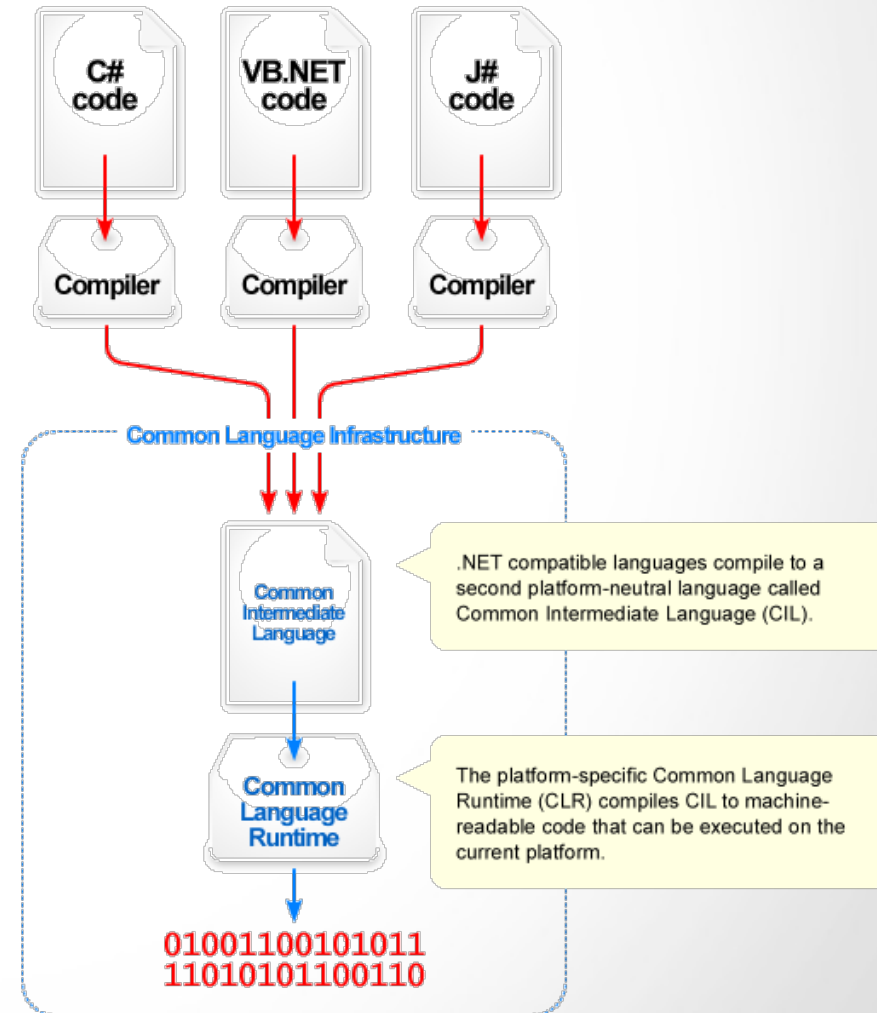
Common Language pour le CLI/CLR chez microsoft

- ✓ **CLI : Common Language Infrastructure**
 - l'environnement d'exécution (donc pas seulement la machine virtuelle) basée sur les deux composants suivants:
- ✓ **CLR: Common Language Runtime**
 - la machine virtuelle chargée d'exécutée le pseudo-code
- ✓ **CIL : Common Intermediate Language**
 - Equivalent bytecode Java
 - Normalisé par l'ECMA
 - Langage proche de la machine
 - MSIL : implémentation MS du CIL



Le CLR et le langage CIL

- ✓ **Compilation du code CIL en langage machine, à l'exécution**
- ✓ **Chaque méthode est compilée juste avant sa première utilisation : Compilation JIT (Just In Time = Juste à temps)**
- ✓ **La compilation JIT est quasi-transparente au niveau des performances car le langage CIL proche du langage machine**
- ✓ **Compilation JIT : permet d'exécuter un même assemblage sur plusieurs types de machines**
- ✓ **Possibilité de précompiler le code CIL d'un assemblage pour un type de machine**





Comment rendre du Code Natif et du p-code interopérable ?

ByteCode en Java et Librairies Natives
Avec Java Native Interface (JNI)



Pourquoi JNI

- ✓ Applications existantes dans un environnement Java, avec ou sans les sources...
- ✓ Programmation d'un nouveau périphérique, logiciel de base, Entrées/Sorties, Cartes d'acquisition, de commandes
 - Adressage physique, Accès au matériel, aux pilotes de la carte, interruptions...
- ✓ Développement en C/C++, tout en bénéficiant de l'environnement Java pour des IHM par exemple
- ✓ Code natif pour de meilleures performances en temps d'exécution (plus toujours vrai)
- ✓ Mais portabilité remise en question



Possibilités de JNI

- ✓ L'API JNI offre l'accès à la machine virtuelle et son environnement
 - accès aux variables d'instance, appel de méthodes, chargement d'une classe, création d'instances...
 - Mécanisme de bas-niveau...
 - Exceptions,
 - Threads....



Deux grands Intérêts ...

- ✓ **Quand Java appelle C/C++ :**
 - Prototypes et Conventions entre les deux langages
 - Chargement dynamique de bibliothèques en Java

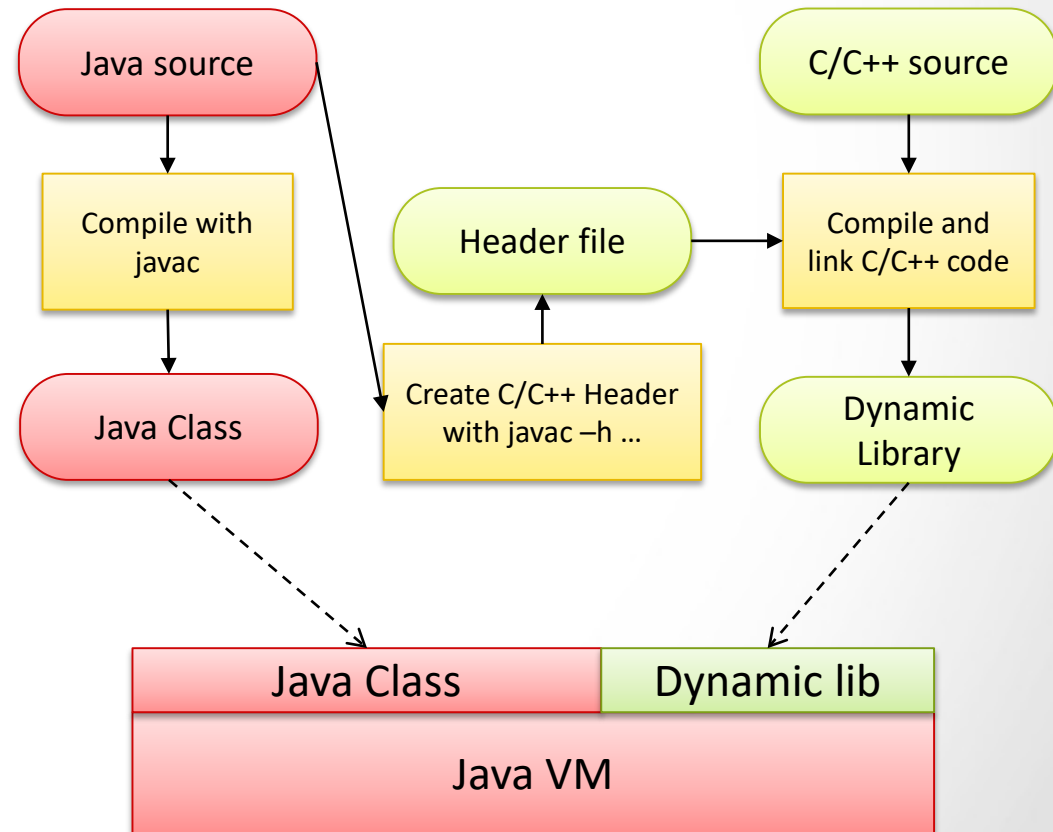
- ✓ **Quand du C/C++ accède à l'environnement Java**
 - lecture/écriture de données d'instance et de classes
 - invocation de méthodes d'instance et de classes
 - création d'objet
 - création de tableaux et de String
 - Levée et filtrage d'exceptions
 - utilisation des moniteurs (de Hoare)
 - Entrées/sorties Série
 - création de machine(s) Java



Quand Java appelle C/C++



1. **Ecrire le code source Java**
2. **Produire le .h (javac -h)**
3. **Ecrire le code source C/C++**
4. **Générer la bibliothèque dynamique**
5. **Exécuter le code java (après avoir rendu la bibliothèque dynamique accessible)**





Compilation avec javac

1. `javac JavaVersC.java`

2. **Usage du mot clé `native`**

3. **Chargement de la bibliothèque (dll/so), dans laquelle sera implémentée le code C de `bonjour`, avec la méthode `loadLibrary` de la classe `System`**

```
public class JavaVersC {  
  
    // méthode statique loadLibrary()  
    // de la classe system  
    static {  
        System.loadLibrary("JavaVersC");  
    }  
  
    public native void bonjour();  
  
    public static void main(String args[]){  
        new JavaVersC().bonjour();  
    }  
}
```

Création du fichier d'entête .h avec javac -h ...

4. Génération de l'interface « .h »

```
javac -h dir JavaVersC.java
```

```
/* file generated */
#include <jni.h>
#ifdef _Included_JavaVersC
#define _Included_JavaVersC
#ifdef __cplusplus
extern "C" {
#endif
/* Class:      JavaVersC,  Method:
   bonjour,    Signature:  ()V
   */
JNIEXPORT void JNICALL
    Java_JavaVersC_bonjour(JNIEnv *,
        jobject);
#ifdef __cplusplus
}
#endif
#endif
```

Compilation et édition de lien du source C/C++

```
#include <stdio.h>
#include "JavaVersC.h"

JNIEXPORT void JNICALL Java_JavaVersC_bonjour (JNIEnv *env, jobject j){
    printf("Java_JavaVersC_bonjour");
}
```

5. Génération de la dll, (JavaVersC.dll)

Avec visual c++

```
cl -Ic:\jdk\include -Ic:\jdk\include\win32 -LD
JavaVersC.c -FeJavaVersC.dll
```

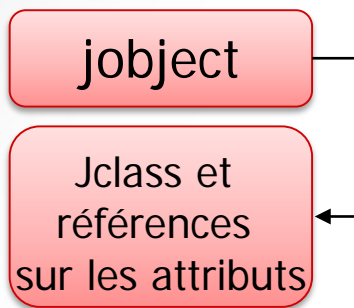
6. Exécution par

```
java JavaVersC
```



JNIEnv et jobject

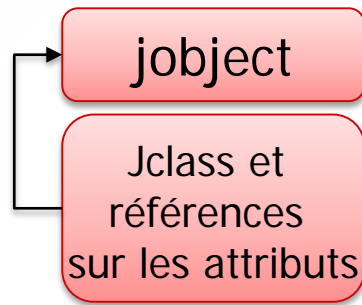
- ✓ JNIEnv *env
 - Il s'agit de l'environnement de la machine Java associé au « Thread » courant, (le Thread ayant engendré l'appel de la méthode native bonjour)
- ✓ jobject j
 - Il s'agit de l'objet receveur du message bonjour(), ici l'instance créée dans la méthode main
- ✓ En résumé
 - A chaque appel d'une méthode native sont transmis par la machine Java
 - un environnement
 - l'objet receveur ou la classe si c'est une méthode static (de classe)
 - et éventuellement les paramètres



Jobject et Jclass

- ✓ L'appel d'une méthode se fait à partir d'un Jobject dès lors que la méthode n'est pas static
- ✓ **Exemple** : `JNIEXPORT void JNICALL Java_Exemple_Test (JNIEnv *env, jobject obj, jint val)` **pour la méthode native** `test(int val)`
- ✓ Néanmoins l'accès aux attributs de cet objet jobject depuis le C/C++ (notamment pour implémenter `Java_Exemple_Test` se basera en fait sur la déclaration des attributs de la classe jclass

Manipulation
des attributs
sur l'objet



Jobject et Jclass

- ✓ La classe jclass est alors obtenue avec :

```
jclass classe = (*env)->GetObjectClass(env, obj);
```

- ✓ Les attributs sont récupérés dans la classe jclass

- ✓ Exemple pour une méthode :

```
jMethodID mid = (*env)->GetMethodID(env, classe, "p", "( )V");
```

- ✓ Ils seront ensuite utilisés en faisant référence à un objet jobject

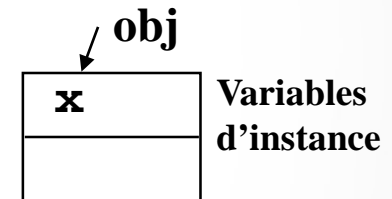
- ✓ Exemple pour la méthode **mid**

```
(*env)->CallVoidMethod(obj, mid, val)
```

Autre exemple : Accès aux Variables d'instance

✓ **GetFieldID, Get<type>Field, Set<type>Field**

```
public class Exemple{  
    private int x;  
    public native void setX(int val);  
}
```



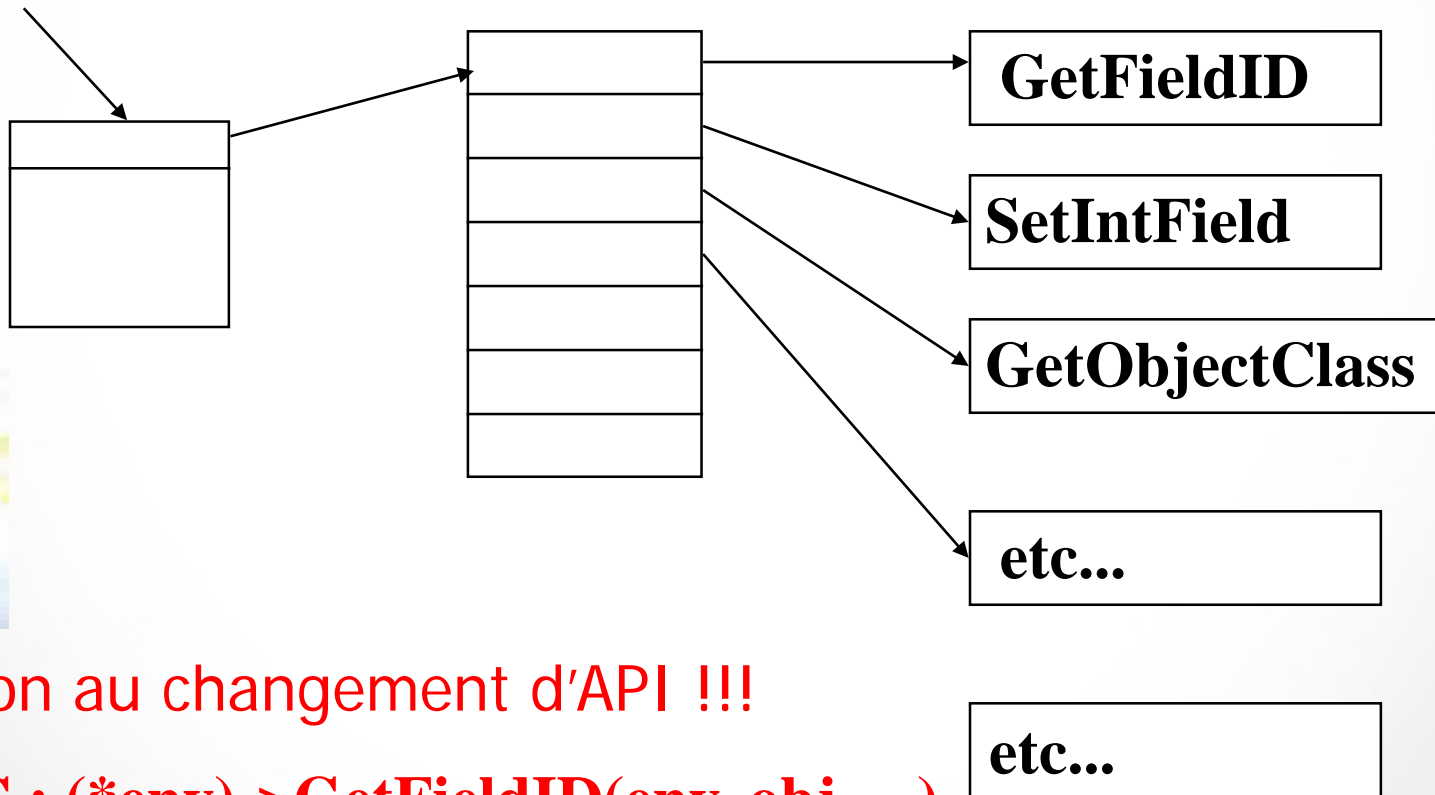
✓ **En C :**

```
JNIEXPORT void JNICALL Java_Exemple_setX (JNIEnv *env,  
jobject obj, jint val){  
    jclass classe = (*env)->GetObjectClass(env,obj);  
    jfieldID fid = (*env)->GetFieldID(env,classe,"x","I");  
    (*env)->SetIntField(env,obj,fid,val) ;  
}
```

✓ **Instructions JVM :** getfield, putfield

Attention JNI API et C ou C++ pour l'accès à `JNIENV *env`

env



Attention au changement d'API !!!

en C : `(*env)->GetFieldID(env, obj, ...)`

en C++ : `env->GetFieldID(obj, ...)`



Mise en œuvre (Java)

```
public class Exemple{
    private int x;

    public native void setX(int val);

    static{
        System.loadLibrary("Exemple");
    }

    public static void main(String args[]) {
        Exemple e = new Exemple();
        e.setX(33);
        System.out.println(" dites " + e.x);
    }
}
```



Mise en œuvre (en C)

```
#include "Exemple.h"
```

```
JNIEXPORT void JNICALL Java_Exemple_setX(JNIEnv* env,  
jobject obj, jint val){  
    jclass cl = (*env)->GetObjectClass(env,obj);  
    jfieldID fid = (*env)->GetFieldID(env, cl, "x", "I");  
    (*env)->SetIntField(env, obj, fid, val) ;  
}
```

Mise en œuvre (en C++)

```
#include "Exemple.h"
```

```
JNIEXPORT void JNICALL Java_Exemple_setX(JNIEnv* env,  
jobject obj, jint val){  
    jclass cl = env->GetObjectClass(obj);  
    jfieldID fid = env->GetFieldID(cl, "x", "I");  
    env->SetIntField(obj, fid, val) ;  
}
```

Les commandes Win32 pour la mise en oeuvre

- ✓ `javac -classpath . Exemple.java`
- ✓ `javac -h dir -classpath . Exemple.java`
- ✓ `cl /Ic:\jdk\include /Ic:\jdk\include\win32
/LD Exemple.c /FeExemple.dll`
- ✓ `java -cp . Exemple`

C/C++ peut aussi accéder à l'environnement du programme Java

- ✓ depuis une dll/so engendrée par `javac -h`
 - emploi du mot clé **native**
 - `java —native——>C/C++`
|
 - `java <—API_JNI—— C/C++`
- ✓ ou depuis une application C/C++ ordinaire quand un exécutable natif lance une machine java et accède à son environnement
 - `java <—API_JNI—— C/C++`



Appel de la machine Java

- ✓ « appel de la JVM et exécution de Java tout en C »
 - utilisation de `javai.lib`
 - chargement et appel de la machine Java depuis le point d'entrée main

- ✓ le source Java ordinaire
 - n'importe quelle application



Exemple (source C++)

```
#include <jni.h>                                /* where everything is defined */
...
JavaVM *jvm;                                  /* denotes a Java VM */
JNIEnv *env;                                  /* pointer to native method interface */
JavaVMInitArgs vm_args = { /* JDK/JRE 10 VM initialization arguments */
    .version = JNI_VERSION_10,
    .nOptions = 0,
    .options = NULL,
    .ignoreUnrecognized = 0,
};

/* load and initialize a Java VM, return a JNI interface pointer in env */
JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);

/* invoke the Main.test method using the JNI */
jclass cls = env->FindClass("Main");
jmethodID mid = env->GetStaticMethodID(cls, "test", "(I)V");
env->CallStaticVoidMethod(cls, mid, 100);

/* We are done. */
jvm->DestroyJavaVM();
```



Annexe : L'API de JNI

Liste des équivalences Code Java / API JNI

Syntaxe des conventions JNI pour les
notations des types et signatures des
méthodes ..



Fonctions de JNI

- Versions
- Opérations sur les classes
- Exceptions
- Références locales et globales
- Opérations sur les objets
- Accès aux champs des objets
- Appels de méthodes d'instances
- Accès aux champs statiques
- Appels de méthodes de classes
- Opérations sur les instances de type String
- Opérations sur les tableaux
- Accès aux moniteurs
- Interface de la JVM

Attention au changement d'API !!!

en C : `(*env)->GetFieldID(env, obj, ...)`

en C++ : `env->GetFieldID(obj, ...)`

Tous les exemples
suivants sont en C



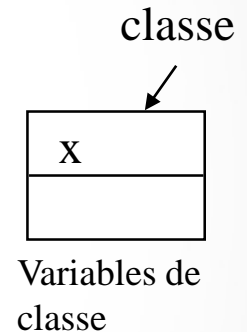
Accès aux Variables de classes

✓ **GetStaticFieldID, GetStatic<type>Field, SetStatic<type>Field**

```
public class Exemple{  
    private static int x;  
    public native setStaticX(int val);  
}
```

```
JNIEXPORT void JNICALL Java_Exemple_setStaticX  
(JNIEnv *env, jobject obj, jint val) {  
    jclass classe = (*env)->GetObjectClass(env, obj);  
    jfieldID fid = (*env)->GetStaticFieldID(env, classe, "x",  
        "I");  
    (*env)->SetStaticIntField(env, classe, fid, val) ;  
}
```

– **instructions JVM** : `getstatic`, `putstatic`

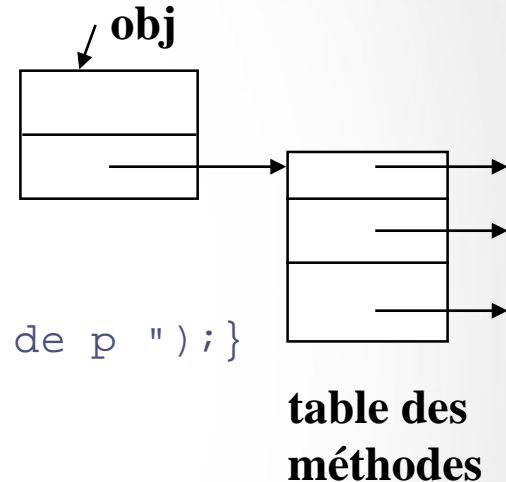




Appels de méthodes d'instance

✓ Call<type>Method

```
public class Exemple{  
    public void p(){System.out.println("appel de p ");}  
    public native callP(int val);  
}
```



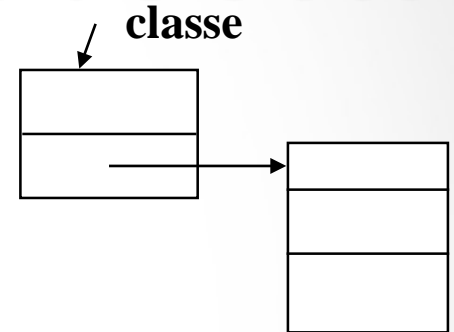
```
JNIEXPORT void JNICALL Java_Exemple_callP (JNIEnv  
*env, jobject obj, jint val){  
    jclass classe = (*env)->GetObjectClass(env, obj);  
    jMethodID mid = (*env)->GetMethodID(env, classe, "p",  
    "()"V");  
    (*env)->CallVoidMethod(env, obj, mid) ;  
}
```

– **instruction JVM** : invokevirtual



Appels de méthodes de classe

✓ `CallStatic<type>Method`



```
public class Exemple{  
    public static void p(){System.out.println( "appel de p ");}  
    public native callP(int val);  
}
```

```
JNIEXPORT void JNICALL Java_Exemple_callP (JNIEnv *env,  
jobject obj){  
    jclass classe = (*env)->GetObjectClass(env, obj);  
    jMethodID mid = (*env)->GetStaticMethodID(env, classe,  
"p", "()V");  
    (*env)->CallStaticVoidMethod(env, classe, mid) ;  
}
```

– **instruction JVM** : `invokestatic`



Types natifs

✓ `jni.h`, `interpreter.h`, `oobj.h`, `typecodes.h`

✓ Types natifs / types java

- `jbyte / byte`, `jshort / short`, `jint / int`

-

- `jobject / Object`, `jclass / Class`, `jstring / String`, `jarray / array`,

- `jthrowable / Throwable`

- **sur la plate-forme Win32**

- **nous avons** `typedef jint long;`



Signature des méthodes

- ✓ **FieldType ::= BaseType | ObjectType | ArrayType**
 - **BaseType**
 - B byte , C char, D double, F float, I int, J long, S short, Z boolean
 - **ObjectType**
 - L<classname>;
 - **ArrayType**
 - [table
- ✓ **MethodDescriptor ::= (FieldType *) ReturnDescriptor**
- ✓ **ReturnDescriptor ::= FieldType | V**
 - V si le type retourné est void

Signature des méthodes en "clair"

✓ javap -s -private JavaVersC

✓ **Compiled from JavaVersC.java**

```
public synchronized class JavaVersC extends
java.lang.Object
    /* ACC_SUPER bit set */
{
    public native void bonjour();
    /*      ()V      */
    public static void main(java.lang.String[]);
    /*      ([Ljava/lang/String;)V      */
    public JavaVersC();
    /*      ()V      */
    static static {};
    /*      ()V      */
}
```



Objets et classes

✓ NewObject, NewObjectA, NewObjectV

– création d'une instance

- obtention de la classe
- obtention du constructeur
- passage des paramètres et création
- ...//en Java : `ClasseA newObj = new ClasseA(10, "hello");`
- ...//en C:

```
jclass classe = (*env)->FindClass(env, "ClasseA");  
jMethodID mid = (*env)->GetMethodID(env, classe,  
"<init>", "(ILjava/lang/String;)V");  
jint val = 10;  
jstring str = (*env)->NewStringUTF(env, "hello");  
jobject newObj = (*env)->NewObject(env, classe, mid,  
val, str) ;}
```

– instruction JVM : `new` et `invokespecial`



✓ IsInstanceOf

```
class A{}  
class B extends A{void callP(boolean b){...};}
```

...// obj est de classe déclarée A mais constatée B

```
jclass classeB = (*env)->FindClass("B");  
if ((*env)->IsInstanceOf(env, obj, classeB)){  
    jMethodID mid = (*env)->GetMethodID(env, classeB, "callP",  
    "(Z)V");  
    jbool val = JNI_TRUE;  
    (*env)->CallVoidMethod(env, obj, mid, val) ;  
}
```

– **instruction JVM** : instanceof, (checkcast)



Tableaux et String

✓ NewObjectArray

```
public class Exemple{  
    public void p(){  
        String sa = newArray(10);  
    }  
    public native String [] newArray(int taille);  
}
```



Tableaux et String

– ...

```
jclass classe = (*env)->FindClass(env, "java/lang/String");
jObjectArray newArr = (*env)->NewObjectArray(env, taille,
classe, NULL);
for(int i = 0; i < taille; i++){
    str = (*env)->NewStringUTF(env, "hello");
    (*env)->SetObjectArrayElement(env, newArr, i, str);
    (*env)->DeleteLocalRef(env, str);
}
return newArr;
```

- **DeleteLocalRef** -> str a 2 références, en jni et en java (gc)

– **instruction JVM** : newarray



Objets et ramasse miettes

- ✓ **Chaque objet crée par JNI ne peut être collecté par le ramasse miettes Java, (l'objet str est référencé dans la machine Java)**
 - `DeleteLocalRef(env, str)` // de l'exemple précédent
 - **permet de libérer cet objet (autorise la récupération mémoire par le ramasse miettes)**
 - `NewGlobalRef(env, obj)`
 - **"bloque" l'objet en mémoire**



Levée d'exceptions

✓ **ThrowNew, ExceptionClear, ExceptionOccured, ExceptionDescribe**

✓ ...

```
jclass classeExc = (*env)->FindClass(env,  
"java/lang/OutOfMemoryError");  
  
...  
if (Condition){  
    (*env)->ThrowNew(env, classeExc, "OutOfMemoryError");  
  
    printf("apres le traitement de l'exception en Java ...");  
    ...  
}
```

✓ **instruction JVM : athrow**



✓ **MonitorEnter, MonitorExit**

```
// l'équivalent de l'instruction synchronized  
(*env)->MonitorEnter(env, obj);
```

```
//du code C
```

```
(*env)->MonitorExit(env, obj);  
}
```

– **instructions JVM** : `monitorenter, monitorexit`



✓ Appels de wait et notify par les primitives GetMethodID et CallVoidMethod

```
jclass classe = (*env)->GetObjectClass(env, obj);
jMethodID waitMid = (*env)->GetMethodID(env, classe, "wait", "()V");
(*env)->CallVoidMethod(env, obj, waitMid);
if((*env)->ExceptionOccured(env) != NULL)
    // une exception est générée en Java, mauvais usage de Wait ...
}
```