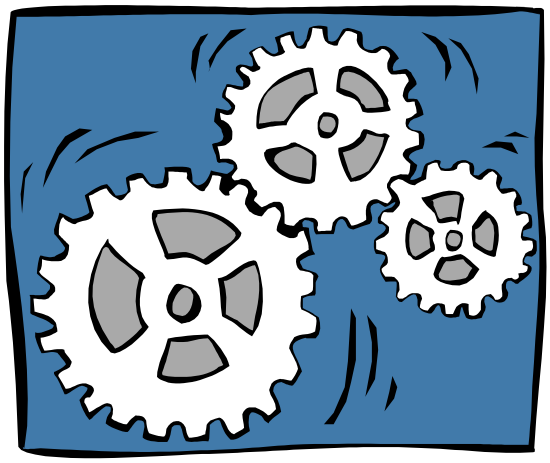

JNI: Java Native Interface

Acerca de este documento

Este documento es un tutorial de JNI (Java Native Interface), el mecanismo que permite ejecutar funciones C y C++ desde Java. El tutorial supone que el lector conoce tanto los lenguajes Java como C, no es necesario conocer C++, aunque el saber siempre ayuda. También supone que el lector está familiarizado con el uso de herramientas de programación como `gcc`, `java` o `javac`. Debido a la interoperatividad del lenguaje Java, hemos pretendido que las explicaciones sean consistentes con cualquier sistema operativo, auque la mayoría de lo ejemplos se desarollan en Mac OS X.

Al acabar este documento esperamos que el lector haya adquirido los conocimientos necesarios para llamar desde Java a librerías C y C++, así como para llamar desde C y C++ a librerías Java.

Nota Legal

Madrid, Octubre del 2007

Para cualquier aclaración contacte con:
`fernando@DELITmacprogramadores.org`

Tabla de Contenido

PARTE I: Programación con JNI

PARTE II: Programación avanzada con JNI

Parte I

Programación con JNI

Sinopsis:

En esta primera parte se tratan los temas más básicos y frecuentes con los que se encuentra un programador Java que está necesitando usar JNI para acceder a librerías nativas. Principalmente se enseña a crear librerías nativas, a utilizar los distintos tipos de variables y objetos Java desde la librería nativa, a llamar desde la librería nativa a métodos Java, y a tratar posibles excepciones. Temas más avanzados se dejan para la segunda parte de este tutorial.

1. Introducción

JNI es un mecanismo que nos permite ejecutar código nativo desde Java y viceversa. El **código nativo** son funciones escritas en un lenguaje de programación como C o C++ para un sistema operativo (a partir de ahora SO) donde se está ejecutando la máquina virtual. Llamamos **máquina virtual** a un entorno de programación formado por:

- El programa que ejecuta los bytecodes Java
- Las librerías (API) de Java

Llamamos **host environment** a tanto al ordenador en que se ejecuta el SO como a las librerías nativas de este ordenador. Estas librerías se suelen escribir en C o C++ y se compilan dando lugar a códigos binarios del hardware donde se ejecuta el host environment. De esta forma podemos decir que las máquinas virtuales Java se hacen para un determinado host environment.

La máquina virtual que nosotros usamos en Mac OS X es el JRE (Java Runtime Environment) de Apple, aunque hay otras máquinas virtuales Java (también llamadas plataformas) como la de Sun que es la implementación de referencia y la más usada en sistemas como Windows, Linux y Solaris. IBM tiene una máquina virtual para Windows, para AIX y para Linux. Microsoft también hizo una máquina virtual para Windows, pero está desestimada por no seguir las especificaciones de Sun.

JNI tiene un interfaz bidireccional que permite a las aplicaciones Java llamar a código nativo y viceversa. La Figura 1 muestra gráficamente esta bidireccionalidad. Es decir JNI soporta dos tipos de interfaces:

1. **Native methods**. Que permiten a Java llamar a funciones implementadas en librerías nativas
2. **Invocation Interface.** que nos permite incrustar una máquina virtual Java en una aplicación nativa. P.e. una máquina virtual Java en un navegador web escrito en C. Para ello, la aplicación nativa llama a librerías nativas de la máquina virtual y luego usa el llamado **invocation interface** para ejecutar métodos Java el la máquina virtual.

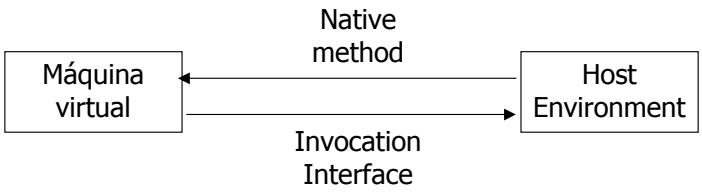

Figura 1: Interfaz bidireccional entre Java y las librerías nativas

2. Evolución histórica de JNI

Desde los orígenes de Java se incluía ya una forma de hacer llamadas nativas desde la máquina virtual a las librerías del host environment, así como desde el host environment a la máquina virtual (en ambas direcciones) pero con 2 problemas:

1. Las llamadas a código nativo desde Java accedían a estructuras C, pero no estaba definida la posición exacta que debían ocupar estos campos en memoria con lo que una llamada nativa en una máquina virtual no coincidía con las llamadas en otra máquina virtual.
2. Las llamadas nativas al JSDK 1.0 se basaban en el uso de un recolector de basura conservativo, ya que no se controlaba cuando ya no quedaba ningún puntero nativo apuntaba a un objeto Java, con lo que una vez apuntado un objeto Java desde un método nativo éste no se liberaba nunca más.

Esta forma de llamadas nativas es lo que se usó en el JSDK 1.0 para implementar clases que accedían a partes cruciales del host como son `java.io.*`, `java.net.*` o `java.awt.*`

Debido a los problemas arriba mencionados, se decidió implementar una nueva forma de acceso a métodos nativos a la que se llamó JNI, y que es una evolución del JRI (Java Runtime Interface), la técnica de acceso nativo de Netscape diseñada por Warren Harris. Con el JSDK 1.1 surgió la primera versión de JNI, pero todavía las clases Java que accedían al host seguían usando la forma de llamada antigua. En JSDK 1.2 se rescribieron todas las clases que accedían directamente al host usando JNI.

3. Las librerías de enlace estático y dinámico

Las **librerías de enlace estático** son ficheros destinados a almacenar funciones, clases y variables globales y tradicionalmente se crean a partir de varios ficheros de código objeto `.o` (UNIX) o `.obj` (Windows). En UNIX las librerías de enlace estático suelen tener la extensión `.a` y en Windows la extensión `.lib`. Mac OS X usa la extensión `.a` de los sistemas UNIX tradicionales. Las funciones de la librería se incluyen dentro del ejecutable durante la fase de enlazado, con lo que una vez generado el ejecutable ya no es necesario disponer de las librerías de enlace estático.

Las **librerías de enlace dinámico** son ficheros cuyas funciones no se incrustan el ejecutable durante la fase de enlazado, sino que en tiempo de ejecución el programa busca el fichero, carga su contenido en memoria y enlaza su contenido según va siendo necesario, es decir según vamos llamando a las funciones. Esto tiene la ventaja de que varios programas pueden compartir las mismas librerías, lo cual reduce el consumo de disco duro, especialmente con las llamadas al SO (APIs) que suelen ser usadas por muchas aplicaciones a la vez. Su extensión también varía dependiendo del SO en que estemos. Extensiones típicas son las que se muestran en la Tabla 1.

Sistema Operativo	**Extensión**
Mac OS X	`.so` o `.dylib`
UNIX	`.so`
Windows	`.dll`

Tabla 1: Extensiones comunes para cada sistema operativo

Para que los programas encuentren las librerías de enlace dinámico, éstas deben ponerse en unos determinados directorios. Cada SO sigue sus reglas. La Tabla 2 resume estas reglas para varios SO.

SO	**Regla de búsqueda de librerías de enlace dinámico**
Mac OS X	Busca en el directorio donde está la aplicación y en las variables de entorno `DYLD_LIBRARY_PATH` y `LD_LIBRARY_PATH`.
UNIX	Busca en los directorios indicados en la variable de entorno `LD_LIBRARY_PATH`[1]
Windows	Busca en el directorio donde esta la aplicación (`.exe`) y en los directorios indicados por la variable de entorno `PATH`

Tabla 2: Reglas de los SO para buscar librerías de enlace dinámico

Para fijar estas variables de entorno existen comandos que varían dependiendo del sistema donde estemos. P.e. si queremos incluir el directorio actual en el path de búsqueda de librerías de enlace dinámico en Mac OS X haríamos:

[1] En algunos sistemas también podemos indicar rutas de búsqueda de librerías en el fichero en el fichero `/etc/ld.so.conf` que es cargado por `/sbin/ldconfig` durante el arranque

```
$ export DYLD_LIBRARY_PATH=$DYLD_LIBRARY_PATH:.
```

En Linux tendríamos que hacer:

```
$ export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
```

Y en Windows haríamos:

```
SET PATH=%PATH%;.
```

La forma de generar una librería de enlace dinámico es casi siempre la misma:

1. Escribimos los ficheros `.c` o `.cpp` con el código fuente
2. Los compilamos dando una opción al compilador que indique que en vez de un ejecutable queremos generar una librería de enlace dinámico.

Esta opción también varía dependiendo del compilador y SO en que estemos. En Mac OS X usaríamos:

```
$ gcc –bundle –I/System/Library/Frameworks/JavaVM.framework/
Versions/A/Headers HolaMundo.c –o libHolaMundo.jnilib
–framework JavaVM
```

En Mac OS X las librerías de enlace dinámico a las que llamamos desde JNI tienen que tener el nombre: `lib`*`Nombre`*`.jnilib`, obsérvese que no se usa la extensión `.dylib` como las librerías de enlace dinámico normales. En Linux usaríamos la opción `-shared` de la forma:

```
$ gcc HolaMundo.c -shared -o libHolaMundo.so
```

En Solaris se usaría el comando:

```
$ cc –G –I/java/include –I/java/include/solaris HolaMundo.c
–o libHolaMundo.so
```

Aquí también los ficheros de librería deben empezar por `lib` y acabar en `.so`. Y en Windows con el compilador en línea de comandos de Microsoft Visual C haríamos:

```
> cl HolaMundo.c /I c:\jdk\include /I c:\jdk\include\win32 /MD /LD
```

`/MD` asegura que la DLL se enlaza con la libraría multihilo C de Win32
`/LD` Pide al compilador generar una librería de enlace dinámico en vez de un ejecutable.

4. El ejemplo básico

Vamos a hacer un programa Java que llama a una función de librería C que dice: "Hola mundo". Para ello vamos a dar los siguientes pasos:

1. Declarar el método nativo como miembro de una clase.

Los métodos nativos llevan el modificador `native` y están sin implementar ya que su implementación estará en una librería nativa, luego hacemos un fichero `HolaMundo.java` como muestra el Listado 1.

```
public class HolaMundo
{
 private native void saluda();
 public static void main(String[] args)
        {
         new HolaMundo().saluda();
        }
 static {
         System.loadLibrary("HolaMundo");
        }
}
```

Listado 1: Programa Java Hola Mundo

Ahora ya podemos compilar la clase con:

```
$ javac HolaMundo.java
```

El método:

```
void <System> loadLibrary(String libraryName)
```

Recibe el nombre de una librería de enlace dinámico y la carga. La librería debe cargarse antes de llamar a cualquier método nativo. El nombre que pasamos en `libraryName` depende de la plataforma donde estemos:

- En Mac OS X el fichero se debería llamar `libHolaMundo.jnilib` y el `libraryName` pasaríamos `"HolaMundo"`
- En Linux o Solaris el fichero se debería llamar `libHolaMundo.so` y el `libraryName` pasaríamos `"HolaMundo"`
- En Windows el fichero se debería llamar `HolaMundo.dll` y el `libraryName` pasaríamos `"HolaMundo"`

Es decir a `loadLibrary()` siempre le pasamos `"HolaMundo"`, aunque el nombre de la librería de enlace dinámico varía en cada plataforma.

2. Crear el fichero de cabecera nativo

Ahora tenemos que generar un fichero `.h` con el prototipo de los métodos nativos a implementar. Para ello usamos el comando:

```
$ javah HolaMundo
```

Siendo `HolaMundo` el nombre de la clase compilada a la que queremos generar el fichero `.h` . El fichero generado tendrá más o menos el siguiente contenido:

```
$ cat HolaMundo.h
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HolaMundo */

#ifndef _Included_HolaMundo
#define _Included_HolaMundo
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HolaMundo
 * Method:    saluda
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HolaMundo_saluda(JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

3. Implementar el método nativo

La implementación de la función nativa debe tener el mismo prototipo que en la cabecera. Por ejemplo podemos crear un fichero `HolaMundo.c` como muestra el Listado 2. Es importante que la extensión del fichero `HolaMundo.c` sea `.c` y no `.cpp`, ya que como veremos en el apartado 6 de la página 86 el puntero `env` está declarado de forma distinta en C que en C++.

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

JNIEXPORT void JNICALL Java_HolaMundo_saluda(JNIEnv* env, jobject
obj)
{
 printf("Hola Mundo\n");
 return;
}
```

Listado 2: Implementación de método nativo

Los parámetros `JNIEnv` y `jobject` se ponen siempre en los métodos nativos, y ya veremos más adelante para qué valen.

4. Compilar el fichero nativo

Usando los comandos que explicamos en el apartado 3 creamos la librería de enlace dinámico. Por ejemplo en Mac OS X haríamos:

```
$ gcc –bundle –I/System/Library/Frameworks/JavaVM.framework/
Versions/A/Headers HolaMundo.c –o libHolaMundo.jnilib
–framework JavaVM
```

5. Ejecutar el programa

Al ejecutar el programa Java se producirá una `UnsatisfiedLinkError` si la función `loadLibrary()` no encuentra la librería. Recuérdese que es necesario fijar las variables de entorno que indican dónde están las librerías de enlace dinámico como explicamos antes. Alternativamente podemos indicar el path de la librería de enlace dinámico con la propiedad del sistema `java.library.path` de la forma:

```
$ java –Djava.library.path=. HolaMundo
Hola Mundo
```

Para que la busque en el directorio actual. Por último comentar que podemos ver las llamadas que se hacen a JNI con la opción `–verbose:jni`

```
$ java -verbose:jni HolaMundo
[Dynamic-linking native method java.lang.StrictMath.sin ... JNI]
[Dynamic-linking native method java.lang.StrictMath.cos ... JNI]
[Dynamic-linking native method java.lang.Float.intBitsToFloat ...
JNI]
[Dynamic-linking native method java.lang.Double.longBitsToDouble ...
JNI]
[Dynamic-linking native method java.lang.Float.floatToIntBits ...
JNI]
[Dynamic-linking native method java.lang.Double.doubleToLongBits ...
JNI]
[Dynamic-linking native method java.lang.Object.registerNatives ...
JNI]
[Registering JNI native method java.lang.Object.hashCode]
[Registering JNI native method java.lang.Object.wait]
[Registering JNI native method java.lang.Object.notify]
[Registering JNI native method java.lang.Object.notifyAll]
[Registering JNI native method java.lang.Object.clone]
[Dynamic-linking native method java.lang.System.registerNatives ...
JNI]
.....................................
.....................................
[Dynamic-linking native method java.io.FileInputStream.available ...
JNI]
[Dynamic-linking native method java.lang.Package.getSystemPackage0
... JNI]
[Dynamic-linking native method java.util.zip.Inflater.reset ... JNI]
[Dynamic-linking native method java.lang.ClassLoader.defineClass0 ...
JNI]
[Dynamic-linking native method HolaMundo.saluda ... JNI]
Hola Mundo
```

5. Tipos de datos fundamentales, arrays y objetos

En este apartado veremos qué correspondencia existe entre los tipos de datos Java y sus correspondientes tipos de datos C. Después veremos cómo intercambiar datos entre Java y C. Empezaremos viendo los tipos de datos fundamentales para luego pasar a estudiar tipos más complejos como los arrays o los objetos.

5.1. Parámetros de un método nativo

Como hemos adelantado en el apartado 4, un método nativo siempre tiene al menos dos parámetros:

```
JNIEXPORT void JNICALL Java_HolaMundo_saluda(JNIEnv* env, jobject
obj);
```

El parámetro `env` apunta a una tabla de punteros a funciones, que son las funciones que usaremos para acceder a los datos Java de JNI. La Figura 2 muestra esta organización.

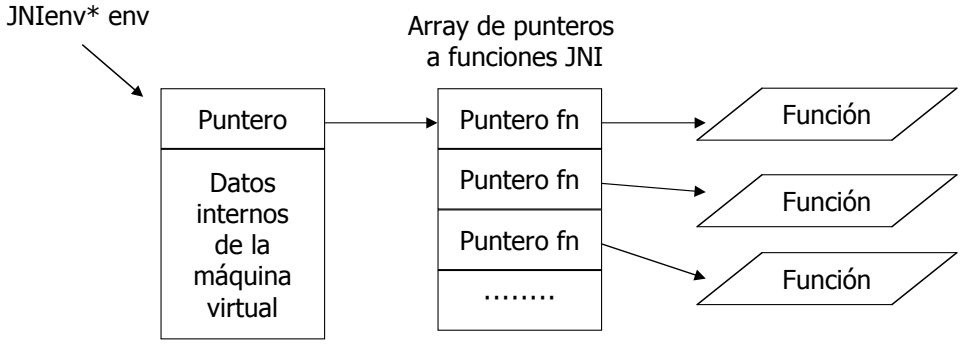

Figura 2: Array de punteros a funciones JNI

El segundo argumento varía dependiendo de si es un método de instancia o un método de clase (estático). Si es un método de instancia se trata de un `jobject` que actúa como un puntero `this` al objeto Java. Si es un método de clase, se trata de una referencia `jclass` a un objeto que representa la clase en la cual están definidos los métodos estáticos.

5.2. Correspondencia de tipos

En Java existen principalmente dos tipos de datos:

Tipos fundamentales. Como `int`, `float` o `double`. Su correspondencia con tipos C es directa, ya que en el fichero `jni.h` encontramos definiciones de tipos C

equivalentes. P.e para `float` encontramos el tipo `jfloat`. La Tabla 3 muestra todos los tipos fundamentales Java y sus correspondientes tipos C.

Tipo Java	Tipo C	Descripción
`boolean`	`jboolean`	8 bits sin signo
`byte`	`jbyte`	8 bits con signo
`char`	`jchar`	16 bits sin signo
`short`	`jshort`	16 bits con signo
`int`	`jint`	32 bits con signo
`long`	`jlong`	64 bits con signo
`float`	`jfloat`	32 bits formato IEEE
`double`	`jdouble`	64 bits formato IEEE

Tabla 3: Tipos de datos fundamentales Java y C

Referencias. Apuntan a objetos y arrays. JNI pasa los objetos a los métodos nativos como referencias opacas, es decir como punteros C que apuntan a estructuras internas sólo conocidas por la máquina virtual concreta que estemos usando. Los campos de esta estructura no se dan a conocer al programador, sino que el programador accede a ellos a través de funciones JNI. P.e. El tipo JNI correspondiente a `java.lang.String` es `jstring`. El valor exacto de los campos a los que apunta `jstring` es irrelevante al programador,es decir, éste accede al texto del objeto llamando a la función:

```
const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean*
isCopy);
```

La función devuelve un array de caracteres UTF-8 correspondientes al `jstring` Java (Unicode 16 bits). Más adelante explicaremos para qué vale el parámetro `isCopy`. Todas las referencias en JNI son del tipo `jobject`, aunque por mejorar el control de tipos se han creado tipos derivados como `jstring` o `jobjectArray`, que tienen la siguiente jerarquía:

```
jobject
      jclass
      jstring
      jarray
            jobjectArray
            jbooleanArray
            jbyteArray
            jcharArray
            jshortArray
            jintArray
            jlongArray
            jfloatArray
            jdoubleArray
jthrowable
```

5.3. Acceso a tipos de datos fundamentales

La conversión entre variables Java y tipos de datos fundamentales es como ya dijimos inmediata. P.e. si queremos hacer un método nativo que sume dos números,

primero crearíamos un programa Java en el fichero `Parametros.java` como el del Listado 3.

```
public class Parametros
{
 private native int suma(int a, int b);
 public static void main(String[] args)
        {
         Parametros p = new Parametros();
         System.out.println("3+4="+p.suma(3,4));
        }
 static{
        System.loadLibrary("Parametros");
       }
}
```

Listado 3: Programa Java que suma dos números

Donde el tipo `int` de Java corresponde con el tipo `jint` de C, y luego tendremos que hacernos un fichero `Parametros.c` como muestra el Listado 4.

```
#include <jni.h>
#include "Parametros.h"

JNIEXPORT jint JNICALL Java_Parametros_suma(JNIEnv * env, jobject
obj, jint a , jint b)
{
 return a+b;
}
```

Listado 4: Función nativa que suma dos números

El prototipo exacto de la función nativa lo encontramos en el fichero `Parametros.h` tras ejecutar `javah`.

5.4. Acceso a objetos `String`

`String` es una clase que está representado por el tipo C `jstring`. Para acceder al contenido de este objeto existen funciones que convierten un `String` Java en cadenas C. Estás funciones nos permiten convertir tanto a Unicode como a UTF-8

5.4.1 Obtener el texto de un `String`

Para obtener el texto UTF-8 correspondiente a un `String` Java tenemos la función:

```
const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean*
isCopy);
```

Que nos devuelve un puntero a un array de caracteres UTF-8 del string. Luego para llamar a esta función haríamos:

```
const jbyte* str = (*env)->GetStringUTFChars(env,text,NULL);
if (str==NULL)
   return NULL; // OutOfMemoryError fue lanzada
```

Donde la variable `text` es un `jstring` que hemos recibido como parámetro desde Java. Como veremos más tarde, el lanzamiento de excepciones en JNI es distinto al lanzamiento de excepciones en Java. Cuando se lanza una excepción en JNI, no se retrocede por la pila de llamadas hasta encontrar el `catch`, sino que la excepción queda pendiente, y en cuando salimos del método nativo es cuando se lanza la excepción. El buffer obtenido por esta llamada no se libera hasta que lo liberamos explícitamente usando la función:

```
void ReleaseStringUTFChars(JNIEnv* env, jstring string, const char*
utf_buffer);
```

Además de las funciones `GetStringUTFChars()` y `ReleaseStringUTFChars()` tenemos las funciones:

```
const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean*
isCopy);
void RealeaseStringChars(JNIEnv* env, jstring string, const jchar*
chars_buffer);
```

Que nos sirven para obtener un buffer con la representación Unicode del texto del `jstring`. Las cadenas UTF-8 siempre acaban en `'\0'`, pero no las cadenas Unicode, con lo que tendremos que usar la función:

```
jsize GetStringLength(JNIEnv* env, jstring string);
```

Para saber la longitud de una cadena Unicode. También podemos preguntar la longitud de una cadena UTF-8 usando:

```
jsize GetStringUTFLength(JNIEnv* env, jstring string);
```

Obsérvese que tanto `GetStringUTFChars()` como `GetStringChars()`tienen un argumento `isCopy`, que indica si el buffer devuelto es una copia del buffer donde está el string de la máquina virtual, o es el mismo string, en cuyo caso nunca deberíamos modificar este buffer o modificaríamos realmente el `String` java, lo cual violaría el principio de inmutabilidad de los `String` Java. En general no es posible predecir si la máquina virtual hará copia del buffer o no.

En principio si la máquina virtual utiliza un formato para almacenar los strings, p.e. Unicode, entonces sabemos que `GetStringUTFChars()` siempre devolverá una copia, y `GetStringChars()` devolverá una copia o no. El devolver una copia implica hacer una copia del buffer, pero a cambio la máquina virtual podrá ejecutar la recogida de basura. Si devuelve el buffer original no podrá realizar recogida de basura del objeto `String` de Java aunque en Java ya nadie lo apunte, porque lo estamos apuntando desde C. El parámetro `isCopy` puede ser `NULL` en caso contrario nos devolverá si ha hecho una copia en otro array o ese array el buffer original. `isCopy` valdrá `JNI_TRUE` o `JNI_FALSE`. Como normalmente un `String` lo cogeremos, procesaremos y liberaremos rápidamente, normalmente es mejor que se haga una copia de éste.

En el JSDK 1.2 surgieron nuevas funciones que, no nos garantizan, pero si aumentan la probabilidad de obtener directamente un puntero al array de caracteres de la máquina virtual:

```
const jchar* GetStringCritical(JNIEnv* env, jstring string, jboolean*
isCopy);
```

```
void ReleaseStringCritical(JNIEnv* env, jstring string, jchar*
char_array);
```

A cambio debemos de tratar el código entre las llamadas a `GetStringCritical()` y `ReleaseStringCritical()` como una sección crítica, respecto a que no debemos de hacer nada que bloquee al hilo, o llamar a otra función de JNI, sólo podemos hacer cosas como copiar el contenido del buffer a otro buffer.

No existen las correspondientes `GetString`**`UTF`**`Critical()` y `ReleaseString`**`UTF`**`Critical()` ya que las máquinas virtuales suelen representar internamente los string en Unicode, y en caso de existir seguramente tendrían que hacer una copia para obtener su representación UTF-8.

5.4.2 Crear un nuevo `String`

También podemos crear nuevas instancias de un `java.lang.String` desde un método nativo usando las funciones JNI:

```
jstring NewStringUTF(JNIEnv* env,const char* bytes);
```

Esta función recibe un array de caracteres UTF-8 y crea un objeto `jstring`

```
jstring NewString(JNIEnv* env, const jchar* ubuffer, jsize length);
```

Esta otra función recibe un array de caracteres Unicode y crea un `jstring`. Ambas funciones producen una excepción `OutOfMemoryError` si fallan en cuyo caso además retornan `NULL`, con lo que siempre hay que comprobar el retorno de la función.

5.4.3 Ejemplo

Como ejemplo proponemos hacer un método nativo que recibe como parámetro un `String` con un mensaje de prompt a dar a un usuario por consola y recoge de consola el texto escrito por el usuario que nos lo devuelve como un `String`. El Listado 5 muestra la forma de la clase Java.

```
public class Parametros
{
 private native int suma(int a, int b);
 private native String pideTexto(String prompt);
 public static void main(String[] args)
        {
         Parametros p = new Parametros();
         String texto = p.pideTexto("Escriba un texto");
         System.out.println("El texto devuelto es:"+texto);
        }
 static{
        System.loadLibrary("Parametros");
       }
}
```

Listado 5: Clase Java que envía y recibe un `String`

La implementación del método nativo en el fichero `Parametros.c` se muestrá en el Listado 6.

```
JNIEXPORT jstring JNICALL Java_Parametros_pideTexto(JNIEnv * env,
jobject obj, jstring prompt)
{
 char buffer[128];
 // Obtenemos el texto pasado como parametro
 const jbyte* prompt_c = (*env)->GetStringUTFChars(env
                                          ,prompt,NULL);
 printf("%s:",prompt_c);
 (*env)->ReleaseStringUTFChars(env,prompt,prompt_c);
 // Pedimos un texto por consola
 // Suponemos que no escribira el usuario más
 // de 127 caracteres
 scanf("%s",buffer);
 return (*env)->NewStringUTF(env,buffer);
}
```

Listado 6: Función nativa que recibe y devuelve un `String` Java

5.5. Acceso a arrays

JNI permite trabajar con dos tipos de arrays:

- Arrays de tipos de datos fundamentales. Son arrays cuyos elementos son tipos de datos fundamentales como `boolean` o `int`.
- Arrays de referencias. Son arrays cuyos elementos son objetos o bien otros arrays (arrays de arrays).

Por ejemplo:

```
int[] iarr; // Array de tipos fundamentales
float[] farr; // Array de tipos fundamentales
Object[] oarr; // Array de referencias
int[][] iarr2; // Array de referencias
```

5.5.1 Acceso a arrays de tipos de datos fundamentales

Para acceder a arrays de tipos fundamentales existen funciones JNI que nos devuelven el array en una variable de tipo `jarray` o derivada. En concreto, existen tipos derivados para los arrays para los tipos de elementos más comunes.

```
jarray
      jobjectArray
      jbooleanArray
      jbyteArray
      jcharArray
      jshortArray
      jintArray
      jlongArray
      jfloatArray
      jdoubleArray
```

Una vez tengamos una variable nativa de tipo `jarray` o derivado podemos obtener la longitud del array con:

```
jsize GetArrayLength(JNIEnv* env, jarray array);
```

Después podemos acceder a los datos del array usando funciones JNI. Para cada tipo fundamental existe su correspondiente función JNI.

```
jboolean* GetBooleanArrayElements(JNIEnv* env, jbooleanArray array,
jboolean* isCopy);
jbyte* GetByteArrayElements(JNIEnv* env, jbyteArray array, jbyte*
isCopy);
jchar* GetCharArrayElements(JNIEnv* env, jcharArray array, jchar*
isCopy);
jshort* GetShortArrayElements(JNIEnv* env, jshortArray array, jshort*
isCopy);
jint* GetIntArrayElements(JNIEnv* env, jintArray array, jint*
isCopy);
jlong* GetLongArrayElements(JNIEnv* env, jlongArray array, jlong*
isCopy);
jfloat* GetFloatArrayElements(JNIEnv* env, jfloatArray array, jfloat*
isCopy);
jdouble* GetDoubleArrayElements(JNIEnv* env, jdoubleArray array,
jdouble* isCopy);
```

El parámetro `isCopy` es un parámetro de salida que nos dice si el puntero devuelto es al array original, o si se ha hecho una copia de éste. Una vez que estas funciones nos devuelve un puntero, éste es válido hasta que lo liberamos con su correspondiente función:

```
void Release<i>Tipo</i>ArrayElements(JNIEnv* env, jbooleanArray array,
jboolean* buffer, jint mode);
```

De la cual también hay una para cada tipo de dato fundamental. El parámetro `mode` puede tomar uno de los valores de la Tabla 4.

Modo	**Acción**
`0`	Copia el contenido de `buffer` en el array Java y libera `buffer`
`JNI_COMMIT`	Copia el contenido de `buffer` en el array Java pero no libera `buffer`
`JNI_ABORT`	Libera `buffer` sin copiar su contenido en el array Java

Tabla 4: Valores de retorno en el parámetro `mode`

En JDK 1.2 se añadieron las funciones JNI:

```
void* GetPrimitiveArrayCritical(JNIEnv* env, jarray array, jboolean*
isCopy);
void ReleasePrimitiveArrayCritical(JNIEnv* env, jarray array, void*
carray, jint mode);
```

Que al igual que pasaba con `String` las funciones procuran enviar el array original si es posible, con lo que el código nativo no debe bloquearse o llamar a otras funciones JNI entre las llamadas a estas funciones. Por último también existe las siguientes funciones para crear arrays Java desde el código nativo:

```
jbooleanArray NewBooleanArray(JNIEnv* env, jsize length);
```

```
jbyteArray NewByteArray(JNIEnv* env, jsize length);
jcharArray NewCharArray(JNIEnv* env, jsize length);
jshortArray NewShortArray(JNIEnv* env, jsize length);
jintArray NewIntArray(JNIEnv* env, jsize length);
jlongArray NewLongArray(JNIEnv* env, jsize length);
jfloatArray NewFloatArray(JNIEnv* env, jsize length);
jdoubleArray NewDoubleArray(JNIEnv* env, jsize length);
```

Como ejemplo podríamos hacer un método que recibe dos arrays unidimensionales y devuelve un nuevo array con la suma de estos. Para empezar, tal como muestra el Listado 7, en la clase Java metemos el método nativo:

```
public class Parametros
{
 private native int suma(int a, int b);
 private native String pideTexto(String prompt);
 private native int[] sumaArrays(int[]A, int[] B);
 public static void main(String[] args)
        {
         Parametros p = new Parametros();
         System.out.println("3+4="+p.suma(3,4));
         String texto = p.pideTexto("Escriba un texto");
         System.out.println("El texto devuelto es:"+texto);
         int[] A = {1,2,3};
         int[] B = {3,2,1};
         int[] C =p.sumaArrays(A,B);
         if (C==null)
            System.out.println("No se pudo hacer la suma");
         else
            {
             System.out.print("La suma de arrays es:");
             for (int i=0;i<C.length;i++)
                 System.out.print(C[i]+" ");
             System.out.println();
            }
        }
 static{
        System.loadLibrary("Parametros");
       }
}
```

Listado 7: Clase Java que pasa y arrays a un método nativo

Y la función C a implementar se muestra en el Listado 8.

```
JNIEXPORT jintArray JNICALL Java_Parametros_sumaArrays
  (JNIEnv * env, jobject obj, jintArray arrayA, jintArray arrayB)
{
 jintArray arrayC;
 jint* A;
 jint* B;
 jint* C;
 jsize i;
 // Medimos los arrays
 jsize longitud = (*env)->GetArrayLength(env,arrayA);
 if (longitud!=(*env)->GetArrayLength(env,arrayB))
    return NULL; // No coinciden las longitudes
 // Creamos un nuevo array con la solucion
 arrayC = (*env)->NewIntArray(env,longitud);
```

```
 A = (jint*)(*env)->GetPrimitiveArrayCritical(env
                                          ,arrayA,NULL);
 B = (jint*)(*env)->GetPrimitiveArrayCritical(env
                                          ,arrayB,NULL);
 C = (jint*)(*env)->GetPrimitiveArrayCritical(env
                                          ,arrayC,NULL);
 for (i=0;i<longitud;i++)
     C[i] = A[i]+B[i];
 (*env)->ReleasePrimitiveArrayCritical(env
                                  ,arrayA,A,JNI_ABORT);
 (*env)->ReleasePrimitiveArrayCritical(env
                                  ,arrayB,B,JNI_ABORT);
 (*env)->ReleasePrimitiveArrayCritical(env,arrayC,C,0);
 return arrayC;
}
```

Listado 8: Función nativa que trabaja con arrays

5.5.2 Acceso a arrays de referencias

JNI tiene dos funciones para el acceso a los arrays de referencias:

```
jobject GetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize
index);
void SetObjectArrayElement(JNIEnv* env, jobjectArray array, jsize
index, jobject value);
```

A diferencia de los arrays de tipos de datos fundamentales, no podemos acceder a todos los elementos a la vez, sino que tenemos que procesar elemento a elemento con las siguientes funciones. Para crear un array de referencias tenemos la función:

```
jarray NewObjectArray(JNIEnv* env, jsize length, jclass elementType,
jobject initialElement);
```

El parámetro `initialElement` es el valor inicial al que se fijan todos los elementos del array, y puede ser `NULL`. El parámetro `elementType` indica el tipo de los elementos del array, y no puede ser `NULL`. Para obtener este tipo se suele usar la función:

```
jclass FindClass(JNIEnv* env, const char* name);
```

Ya comentaremos cómo funciona esta función, de momento basta decir que para el ejemplo que vamos a hacer en `name` pasaremos `"[I"` que significa que queremos un array de arrays de enteros.

Como ejemplo vamos ha hacer un método `sumaMatrices()`, que recibe dos matrices bidimensionales y devuelve otra matriz con la suma de estas. El código Java se muestra en el Listado 9.

```
public class Parametros
{
 private native String pideTexto(String prompt);
 private native int[] sumaArrays(int[]A, int[] B);
 private native int[][] sumaMatrices(int[][] A, int[][]B);
 public static void main(String[] args)
        {
         Parametros p = new Parametros();
```

```
         String texto = p.pideTexto("Escriba un texto");
         System.out.println("El texto devuelto es:"+texto);
         int[] A = {1,2,3};
         int[] B = {3,2,1};
         int[] C =p.sumaArrays(A,B);
         if (C==null)
            System.out.println("No se pudo hacer la suma");
         else
            {
             System.out.print("La suma de arrays es:");
             for (int i=0;i<C.length;i++)
                 System.out.print(C[i]+" ");
             System.out.println();
            }
         int[][] M1 = {{2,3},{4,6},{2,4}};
         int[][] M2 = {{4,3},{0,2},{1,5}};
         int[][] M3 = p.sumaMatrices(M1,M2);
         if (M3==null)
            System.out.println("Fallo la suma");
         else
            {
             System.out.println("La suma es:");
             for (int i=0;i<M3.length;i++)
                 {
                  for (int j=0;j<M3[i].length;j++)
                      System.out.print(M3[i][j]+" ");
                  System.out.println();
                 }
            }
        }
 static{
        System.loadLibrary("Parametros");
       }
}
```

Listado 9: Programa Java que llama a función nativa para sumar matrices

La implementación de la función nativa se muestra en el Listado 10.

```
JNIEXPORT jobjectArray JNICALL Java_Parametros_sumaMatrices
  (JNIEnv *env, jobject obj, jobjectArray arrayA, jobjectArray
arrayB)
{
 jobjectArray arrayC;
 jsize i;
 jclass tipo_array;
 jsize n_vectores = (*env)->GetArrayLength(env,arrayA);
 if (n_vectores!=(*env)->GetArrayLength(env,arrayB))
    return NULL; // No se pueden sumar los arrays
                 // por ser distintos
 tipo_array = (*env)->FindClass(env,"[I");
 if (tipo_array==NULL)
    return NULL; // Se produjo excepcion
 arrayC = (*env)->NewObjectArray(env,n_vectores,tipo_array,NULL);
 if (arrayC==NULL)
    return NULL; // Excepcion OutOfMemoryError lanzada
 for(i=0;i<n_vectores;i++)
    {
     jsize j;
     jint* bufferA;
```

```
     jint* bufferB;
     jint* bufferC;
     jintArray vectorA = (jintArray)
            (*env)->GetObjectArrayElement(env,arrayA,i);
     jintArray vectorB = (jintArray)
            (*env)->GetObjectArrayElement(env,arrayB,i);
     jsize longitud_vector =
            (*env)->GetArrayLength(env,vectorA);
     jintArray vectorC =
               (*env)->NewIntArray(env,longitud_vector);
     bufferA =
          (*env)->GetIntArrayElements(env,vectorA,NULL);
     bufferB =
          (*env)->GetIntArrayElements(env,vectorB,NULL);
     bufferC =
          (*env)->GetIntArrayElements(env,vectorC,NULL);
     for (j=0;j<longitud_vector;j++)
         bufferC[j] = bufferA[j] + bufferB[j];
     (*env)->ReleaseIntArrayElements(env,vectorA,bufferA,JNI_ABORT);
     (*env)->ReleaseIntArrayElements(env,vectorB,bufferB,JNI_ABORT);
     (*env)->ReleaseIntArrayElements(env
                                    ,vectorC,bufferC,0);
     (*env)->SetObjectArrayElement(env,arrayC,i,vectorC);
     (*env)->DeleteLocalRef(env,vectorA);
     (*env)->DeleteLocalRef(env,vectorB);
     (*env)->DeleteLocalRef(env,vectorC);
    }
 return arrayC;
}
```

Listado 10: Función nativa que suma matrices

La llamada a `DeleteLocalRef()` lo que hace es evitar que la máquina virtual pueda quedarse sin memoria, como explicaremos más adelante.

6. Acceso a objetos

En este apartado aprenderemos a acceder a los atributos de un objeto Java, así como a llamar a sus métodos.

6.1. La signatura de tipo

Para describir los tipos de los atributos y de los métodos de un objeto, Java utiliza las **signaturas de tipo**, también llamadas **descriptores de miembro** (member descriptor). Una signatura de tipo es una cadena C que describe el tipo de un miembro (atributo o método) de la clase. P.e. `"Ljava/lang/String;"` indica que un atributo es del tipo `java.lang.String`. Los tipos fundamentales se representan con letras tal como se indica en la Tabla 5.

Signatura de tipo	Tipo Java
V	void
Z	boolean
B	byte
C	char
S	short
I	int
J	long
F	float
D	double

Tabla 5: Códigos de signaturas de tipo

P.e para representar un `int` usamos `"I"` y para representar un `float` `"F"`. Para representar un array se pone un `[` delante del tipo. P.e. para representar un `int[]` se usa `"[I"` y para representar un `double[][]` se usa `"[[D"`. Por último para representar una clase se precede por una `L`, se pone el nombre completo de la clase y se acaba en `;`. P.e. para representar la clase `java.util.Date` se usa `"Ljava/util/Date;"`. Obsérvese que además se sustituyen los `.` por `/`.

Los métodos también tienen una signatura de tipo en cuyo caso los parámetros del método se ponen primero entre paréntesis y el retorno del método después. P.e. para un método como:

```
public int getWidth()
```

Tenemos la signatura de tipo: `"()I"`. Aquí hay que tener cuidado de no poner `"(V)I"` que seria incorrecto. Sin embargo si tenemos el método:

```
public void drawLine(int width)
```

Su signatura de tipo sería `"(I)V"`, es decir para indicar el retorno si se pone la `V`. O para el método:

```
public java.lang.String formatDate(java.util.Date)
```

Tenemos la signatura de tipo: `"(Ljava/util/Date;)Ljava.lang.String;"`. Si el método recibe varios parámetros se ponen todos seguidos sin usar ningún símbolo como separador. P.e. para:

```
public int suma(int a, int b)
```

La signatura de tipo del método sería: `"(II)I"`. Podemos sacar las signaturas de tipo de los miembros de una clase usando el comando `javap`:

```
$ javap -s -p Parametros
Compiled from Parametros.java
public class Parametros extends java.lang.Object {
    public Parametros();
        /*   ()V   */
    private native int suma(int, int);
       /*   (II)I   */
    private native java.lang.String pideTexto(java.lang.String);
        /*   (Ljava/lang/String;)Ljava/lang/String;   */
    private native int sumaArrays(int[], int[])[];
        /*   ([I[I)[I   */
    private native int sumaMatrices(int[][], int[][])[][];
        /*   ([[I[[I)[[I   */
    public static void main(java.lang.String[]);
        /*   ([Ljava/lang/String;)V   */
    static {};
        /*   ()V   */
}
```

El parámetro `-s` es para que saque las signaturas de tipo C y no la signatura de tipo Java, y `-p` es para que también saque la signatura de tipo de los miembros privados. Una vez tengamos la signatura de tipo, podemos pasársela a funciones como:

```
jclass FindClass(JNIEnv* env,const char* name);
```

La cual, como vimos en el ejemplo anterior nos devuelve una variable que representa a la clase, y que luego esta variable nos la piden otras funciones de JNI. P.e. cuando hacíamos en el ejemplo anterior:

```
tipo_array = (*env)->FindClass(env,"[I");
```

6.2. Acceso a los atributos de un objeto

Java soporta dos tipos de atributos:

- **Atributos de instancia**, de los cuales existe uno por cada objeto que instanciemos.
- **Atributos de clase**, de los cuales sólo existe uno puesto en la clase, en vez de en los objetos de esa clase. Se distinguen porque llevan el modificador `static`.

6.2.1 Acceso a atributos de instancia

Para acceder a un atributo de instancia se siguen dos pasos:

1. Obtenemos el **field ID** del atributo al que queremos acceder, que es una variable de tipo `jfieldID` usada por JNI para referirse a un atributo de un objeto. Para obtenerlo usamos la función:

```
jfieldID GetFieldID(JNIEnv* env, jclass class, const char* name,
const char* signature);
```

2. Usamos una función que nos permita leer o modificar el atributo a partir de su `jobject` y su `jfieldID`. Existe una función para cada tipo de dato, como se muestra a continuación:

```
jboolean GetBooleanField(JNIEnv* env, jobject object, jfieldID
fieldID);
jbyte GetByteField(JNIEnv* env, jobject object, jfieldID fieldID);
jshort GetShortField(JNIEnv* env, jobject object, jfieldID fieldID);
jchar GetCharField(JNIEnv* env, jobject object, jfieldID fieldID);
jint GetIntField(JNIEnv* env, jobject object, jfieldID fieldID);
jlong GetLongField(JNIEnv* env, jobject object, jfieldID fieldID);
jfloat GetFloatField(JNIEnv* env, jobject object, jfieldID fieldID);
jdouble GetDoubleField(JNIEnv* env, jobject object, jfieldID
fieldID);
```

También existe otra para leer atributos que sean objetos o arrays:

```
jobject GetObjectField(JNIEnv* env, jobject object, jfieldID
fieldID);
```

Y también existen sus correspondientes funciones setter:

```
void SetBooleanField(JNIEnv* env, jobject obj, jfieldID fieldID,
jboolean value);
void SetByteField(JNIEnv* env, jobject obj, jfieldID fieldID, jbyte
value);
void SetShortField(JNIEnv* env, jobject obj, jfieldID fieldID, jshort
value);
void SetCharField(JNIEnv* env, jobject obj, jfieldID fieldID, jchar
value);
void SetIntField(JNIEnv* env, jobject obj, jfieldID fieldID, jint
value);
void SetLongField(JNIEnv* env, jobject obj, jfieldID fieldID, jlong
value);
void SetFloatField(JNIEnv* env, jobject obj, jfieldID fieldID, jfloat
value);
void SetDoubleField(JNIEnv* env, jobject obj, jfieldID fieldID,
jdouble value);
void SetObjectField(JNIEnv* env, jobject obj, jfieldID fieldID,
jobject value);
```

Como ejemplo vamos a hacer un método nativo en una clase que incrementa el valor de un atributo de la clase cada vez que lo llamamos. El Listado 11 muestra la clase Java.

```
public class Miembros
{
 private int atributo=0;
 private native void incrementa();
```

```
 public static void main(String[] args)
     {
     Miembros m = new Miembros();
     System.out.println(
          "Antes de llamar a incrementa() atributo vale:"
          +m.atributo);
     m.incrementa();
     System.out.println(
          "Despues de llamar a incrementa() atributo vale:"
          +m.atributo);
  }
 static{
        System.loadLibrary("Miembros");
       }
}
```

Listado 11: Clase Java con método nativo que accede a un atributo de la clase

Y la implementación del método nativo se muestra en el Listado 12.

```
#include <jni.h>

JNIEXPORT void JNICALL Java_Miembros_incrementa(JNIEnv * env, jobject
obj)
{
 jint valor;
 // Obtenemos una referencia a la clase del objeto
 jclass clase = (*env)->GetObjectClass(env,obj);

 // Obtenemos el jfieldID del atributo
 jfieldID fieldID = (*env)->GetFieldID(env
                            ,clase,"atributo","I");
 if (fieldID==NULL)
    return; // Excepcion lanzada
 valor = (*env)->GetIntField(env,obj,fieldID);
 valor++;
 (*env)->SetIntField(env,obj,fieldID,valor);
}
```

Listado 12: Función que implementa un método nativo que accede a un atributo

Obsérvese que necesitamos obtener una referencia a la clase del objeto, para lo cual hemos usado la función:

```
jclass GetObjectClass(JNIEnv* env, jobject object);
```

6.2.2 Acceso a atributos de clase

El acceso a los atributos de clase es similar al acceso a los atributos de instancia sólo que para obtener el field ID usamos la función:

```
jfiledID GetStaticFieldID(JNIEnv* env, jclass class, cost char* name,
const char* signature);
```

Después para acceder al valor usamos las funciones:

```
jboolean GetStaticBooleanFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jbyte GetStaticByteFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jshort GetStaticShortFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jchar GetStaticCharFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jint GetStaticIntFiled(JNIEnv* env, jclass class, jfieldID fieldID);
jlong GetStaticLongFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jfloat GetStaticFloatFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jdouble GetStaticDoubleFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
jobject GetStaticObjectFiled(JNIEnv* env, jclass class, jfieldID
fieldID);
```

Que en vez de recibir un `jobject` como en el acceso a atributos de instancia, reciben un `jclass`. También existen sus correspondientes funciones setter:

```
void SetStaticBooleanField(JNIEnv* env, jclass class, jfieldID
fieldID, jboolean value);
void SetStaticByteField(JNIEnv* env, jclass class, jfieldID fieldID,
jbyte value);
void SetStaticShortField(JNIEnv* env, jclass class, jfieldID fieldID,
jshort value);
void SetStaticCharField(JNIEnv* env, jclass class, jfieldID fieldID,
jchar value);
void SetStaticIntField(JNIEnv* env, jclass class, jfieldID fieldID,
jnit value);
void SetStaticLongField(JNIEnv* env, jclass class, jfieldID fieldID,
jlong value);
void SetStaticFloatField(JNIEnv* env, jclass class, jfieldID fieldID,
jfloat value);
void SetStaticDoubleField(JNIEnv* env, jclass class, jfieldID
fieldID, jdouble value);
void SetStaticObjectField(JNIEnv* env, jclass class, jfieldID
fieldID, jobject value);
```

6.3. Acceso a los métodos de un objeto

Además de poder acceder a los atributos de un objeto Java desde un método nativo, también podemos acceder desde un método nativo a sus métodos Java, a este acceso muchas veces se le llama **acceso callback**, porque primero Java ejecutó un método nativo, y luego el método nativo vuelve a ejecutar un método Java. La Figura 3 resume gráficamente esta idea.

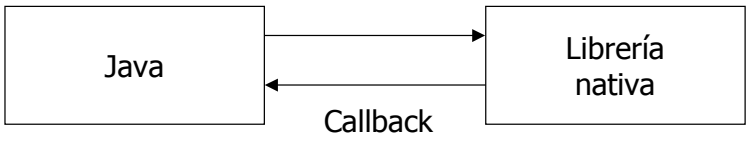

Figura 3: Llamadas callback a métodos Java

Las llamadas callback se hacen de forma distinta dependiendo del tipo del método Java: Métodos de instancia, métodos de clase y constructores. En los siguientes subapartados vamos a comentar cómo se hacen las llamadas callback en cada caso.

6.3.1 Ejecutar métodos de instancia

Para ejecutar un método de instancia de un objeto tenemos que hacer dos cosas:

1. Obtener el **method ID** del método usando:

```
jmethodID GetMethodID(JNIEnv* env, jclass class, const char* name,
const char* signature);
```

2. Ejecutar el método usando una de las siguientes funciones:

```
void CallVoidMethod(JNIEnv* env, jobject object, jmethodID methodID,
...);
jboolean CallBooleanMethod(JNIEnv* env, jobject object, jmethodID
methodID, ...);
jbyte CallByteMethod(JNIEnv* env, jobject object, jmethodID methodID,
...);
jshort CallShortMethod(JNIEnv* env, jobject object, jmethodID
methodID, ...);
jchar CallCharMethod(JNIEnv* env, jobject object, jmethodID methodID,
...);
jint CallIntMethod(JNIEnv* env, jobject object, jmethodID methodID,
...);
jlong CallLongMethod(JNIEnv* env, jobject object, jmethodID methodID,
...);
jfloat CallFloatMethod(JNIEnv* env, jobject object, jmethodID
methodID, ...);
jdouble CallDoubleMethod(JNIEnv* env, jobject object, jmethodID
methodID, ...);
jobject CallObjectMethod(JNIEnv* env, jobject object, jmethodID
methodID, ...);
```

Debemos de usar una función u otra en función del tipo de retorno que tenga el método. Los parámetros se pasan en la lista de parámetros variables que tiene al final la función (...). `CallObjectMethod()` se usa tanto para los métodos que devuelven objetos, como para los que devuelven arrays, los métodos que devuelven tipos fundamentales debemos ejecutarlos usando la función que corresponda de las que se enumeran arriba.

Como ejemplo, en el Listado 14 hemos implementado un método nativo `sumaC()` que llama al método Java `sumaJava()` del Listado 13.

```
public class Miembros
{
 private int atributo=0;
 private native void incrementa();
 private int sumaJava(int a, int b)
        {
         return a+b;
        }
 private native void sumaC();
 public static void main(String[] args)
```

```
        {
         Miembros m = new Miembros();
         System.out.println("Antes de llamar a"
             + " incrementa() atributo vale:"+m.atributo);
         m.incrementa();
         System.out.println("Despues de llamar a"
             + " incrementa() atributo vale:"+m.atributo);
         m.sumaC();
        }
 static{
        System.loadLibrary("Miembros");
       }
}
```

Listado 13: Clase Java con un método callback

```
JNIEXPORT void JNICALL Java_Miembros_sumaC(JNIEnv * env, jobject
object)
{
 int a=5,b=4,c;
 // Obtenemos el method ID
 jclass clase = (*env)->FindClass(env,"Miembros");
 jmethodID methodID =
      (*env)->GetMethodID(env,clase,"sumaJava","(II)I");
 // Ejecutamos el metodo
 c = (*env)->CallIntMethod(env,object,methodID,a,b);
 printf("La suma de %i y %i es %i\n",a,b,c);
}
```

Listado 14: Método nativo que llama a un método de instancia Java

6.3.2 Ejecutar métodos de clase

El proceso de ejecutar métodos de clase es similar al de ejecutar métodos de instancia, sólo que ahora usamos otras funciones. En concreto los pasos son:

1. Para obtener el **method ID** usamos:

```
jmethodID GetStaticMethodID(JNIEnv* env, jclass class, const char*
name, const char* signature);
```

2. Para ejecutar los métodos usamos las funciones:

```
void CallStaticVoidMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jbyte CallStaticByteMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jshort CallStaticShortMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jchar CallStaticCharMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jint CallStaticIntMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jlong CallStaticLongMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jfloat CallStaticFloatMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
```

```
jdouble CallStaticDoubleMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
jobject CallStaticObjectMethod(JNIEnv* env, jclass class, jmethodID
methodID,...);
```

Que son idénticas a las de ejecutar métodos de instancia, sólo que ahora reciben como parámetro un `jclass` en vez de un `jobject`.

6.3.3 Ejecutar métodos de instancia de la superclase

En JNI podemos obtener la clase base de una clase usando:

```
jclass GetSuperClass(HNIEnv* env, jclass class);
```

Que retorna la superclase de una clase o `NULL` si `class` es un `java.lang.Object` o una interface.

Por otro lado, si ejecutamos un método que ha sido redefinido sobre una referencia de tipo base usando las funciones `Call`*`Tipo`*`Method()` se ejecutará el método de la clase derivada, ya que Java usa por defecto enlace dinámico. Si queremos ejecutar el método de la base tenemos que usar las funciones:

```
void CallNonvirtualVoidMethod(JNIEnv* env, jobject obj, jclass class,
jmethodID methodID,...);
jbyte CallNonvirtualByteMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jshort CallNonvirtualShortMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jchar CallNonvirtualCharMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jint CallNonvirtualIntMethod(JNIEnv* env, jobject obj, jclass class,
jmethodID methodID,...);
jlong CallNonvirtualLongMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jfloat CallNonvirtualFloatMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jdouble CallNonvirtualDoubleMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
jobject CallNonvirtualObjectMethod(JNIEnv* env, jobject obj, jclass
class, jmethodID methodID,...);
```

Por ejemplo, en el Listado 15 tenemos las clases Java: Una `Base` y una `Derivada`. Obsérvese que `metodoJava()` está redefinido. El el Listado 16 vamos a probar a llamarlo sobre un objeto de tipo `Derivada`, usando tanto `CallObjectMethod()` como `CallNonvirtualObjectMethod()` para observar la diferencia.

```
public class Base
{
 public String metodoJava()
        {
         return "En la base";
        }
 public native void metodoC();
 public static void main(String[] args)
        {
       Base b = new Derivada();
```

```
      b.metodoC();
     }
 static{
        System.loadLibrary("Base");
       }
}

public class Derivada extends Base
{
 public String metodoJava()
        {
      return "En la derivada";
     }
}
```

Listado 15: Clases Java base y derivada

```
JNIEXPORT void JNICALL Java_Base_metodoC(JNIEnv* env, jobject obj)
{
 // Obtenemos el method ID de Base
 jclass clase = (*env)->FindClass(env,"Base");
 jmethodID methodID = (*env)->GetMethodID(env
            ,clase,"metodoJava","()Ljava/lang/String;");
 // Ejecutamos el metodo sobre obj que es una referencia
 // de tipo Base que apunta a un objeto Derivada
 // y ejecutara el metodo de la Derivada
 jstring mensaje = (jstring)
          (*env)->CallObjectMethod(env,obj,methodID);
 const jbyte* mensaje_c =
          (*env)->GetStringUTFChars(env,mensaje,NULL);
 printf("Al llamar a metodoJava() obtenemos el mensaje:%s\n",
        mensaje_c);
 (*env)->ReleaseStringUTFChars(env,mensaje,mensaje_c);
 // Ahora queremos ejecutar el metodo de Base
 mensaje = (jstring) (*env)->CallNonvirtualObjectMethod(
                                  env,obj,clase,methodID);
 mensaje_c = (*env)->GetStringUTFChars(env,mensaje,NULL);
 printf("Al llamar a metodoJava() obtenemos el mensaje:%s\n",
        mensaje_c);
 (*env)->ReleaseStringUTFChars(env,mensaje,mensaje_c);
}
```

Listado 16: Método nativo que llama a método Java redefinido

La salida que obtenemos es:

```
$ java Base
Al llamar al métodoJava() obtenemos el mensaje:En la derivada
Al llamar al métodoJava() obtenemos el mensaje:En la base
```

Obsérvese que esto es parecido al uso de `super` en Java, pero no igual ya que en Java `super` sólo nos permite llamar a métodos de la clase inmediatamente base, mientras que con `CallNovirtual`*`Tipo`*`Method()` podemos ejecutar métodos de cualquier base.

P.e. si tenemos la jerarquía de clases de la Figura 4, tanto en JNI como en Java podemos llamar desde B al método de A, pero desde C no podemos ejecutar métodos de la clase A usando `super`, pero sí desde JNI.

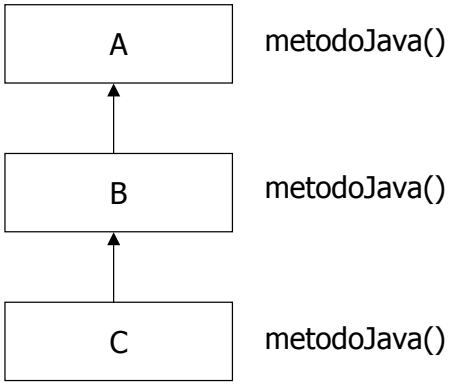

Figura 4: Métodos Java redefinidos

También otra diferencia es que podemos ejecutar métodos de la clase base desde fuera del objeto, cosa que no podemos hacer en Java con el especificador `super`. Es decir en Java no podemos hacer:

```
A a = new B();
a.super.metodo();
```

6.3.4 Invocar a un constructor

Para llamar a un constructor se llama igual que cualquier otro método de instancia, sólo que al ir a acceder al constructor usamos `"<init>"` como nombre del método. Una vez tengamos el method ID podemos pasárselo a la función:

```
jobject NewObject(JNIEnv, jclass class, jmethodID constructorID,...);
```

La cual instancia el objeto y ejecuta sobre el objeto el constructor especificado. Existe otra forma de crear objetos Java, que consiste en usar la función:

```
jobject AllocObject(JNIEnv* env, jclass class);
```

que nos crea un objeto sin inicializar, y luego usamos una función `CallNovirtual`*`Tipo`*`Method()` para ejecutar el constructor sobre el objeto. Esta segunda forma es menos recomendable porque podríamos dejarnos el objeto sin inicializar si se nos olvida llamar a `CallNovirtual`*`Tipo`*`Method()` después de crear el objeto con `AllocObject()`.
Como ejemplo, en el Listado 18 vamos a hacer un método nativo que crea un objeto `Punto` y lo retorna con valores pasados como parámetros al método nativo. Lo vamos a implementar tanto con `NewObject()` como con `AllocObject()`. El Listado 17 muestra la clase Java donde está declarada la clase `Punto` y los métodos nativos.

```
public class Punto
{
 private int x;
 private int y;
 public Punto(int x, int y)
        {
         this.x = x;
```

```
         this.x = y;
        }
 public String toString()
        {
         return "["+x+","+y+"]";
        }
 public static native Punto creaPunto1(int x, int y);
 public static native Punto creaPunto2(int x, int y);
 public static void main(String[] args)
        {
         System.out.println(creaPunto1(2,3));
         System.out.println(creaPunto2(1,5));
        }
 static {
         System.loadLibrary("Punto");
        }
}
```

Listado 17: Clase Java con métodos nativos

```
JNIEXPORT jobject JNICALL Java_Punto_creaPunto1(JNIEnv * env, jclass
clase, jint x, jint y)
{
 jmethodID methodID =
       (*env)->GetMethodID(env,clase,"<init>","(II)V");
 if (methodID==NULL)
    return NULL;
 return (*env)->NewObject(env,clase,methodID,x,y);
}

JNIEXPORT jobject JNICALL Java_Punto_creaPunto2(JNIEnv *env, jclass
clase, jint x , jint y)
{
 jobject obj;
 jmethodID methodID =
        (*env)->GetMethodID(env,clase,"<init>","(II)V");
 if (methodID==NULL)
    return NULL;
 obj = (*env)->AllocObject(env,clase);
 (*env)->CallNonvirtualVoidMethod(env,obj
                            ,clase,methodID,x,y);
 return obj;
}
```

Listado 18: Funciones nativas que instancian objetos java

6.4. Rendimiento en le acceso a arrays, métodos y atributos

Una decisión de diseño importante en JNI es que el acceso a los miembros de una clase se realiza siempre a través de funciones JNI de acceso, en vez de permitir a los métodos nativos un acceso directo. Esto tiene una ventaja y un inconveniente. La ventaja es que los objetos apuntados por referencias se ven como tipos opacos al programador. El inconveniente es que repercute en el rendimiento. Este decremento en el rendimiento se ve atenuado siempre que los métodos nativos no realicen

operaciones triviales, y de hecho los métodos nativos (aparte de en este tutorial) no suelen realizar operaciones triviales.

6.4.1 Rendimiento en el acceso a arrays

El llamar a una función para acceder a cada elemento de un array se vuelve inaceptable cuando vamos a iterar un array de muchos elementos, con lo que se pensó una solución intermedia llamada **pinning**, que consiste en que el método nativo puede pedir a la máquina virtual que no mueva en la memoria el contenido de un array, de esa forma el método nativo puede obtener un puntero que apunta directamente a el contenido de un array. Esta solución tiene dos consecuencias:

- El recolector de basura debe soportar el pinning
- La máquina virtual debe colocar los arrays de tipos de datos fundamentales en posiciones consecutivas de memoria.

Recuérdese que sólo podíamos obtener punteros a arrays de tipos de datos fundamentales con funciones del tipo `Get`*`Tipo`*`ArrayElements()`, a los arrays de objetos tenemos que acceder a cada elemento de forma individual usando `GetObjectArrayElement()`. Aun así `Get`*`Tipo`*`ArrayElements()` no nos garantizaba que obtuviésemos un puntero al array original en vez de hacer una copia en otra zona de memoria, en concreto esta decisión depende de (1) que el recolector de basura soporte pinning, lo cual implica que el objeto no pueda ser liberado aunque no queden referencias Java a él hasta que lo libere JNI con `ReleaseTipoArrayElements()`, y (2) de que la distribución de los elementos del array en memoria, así como el tamaño de las variables sea la misma que la que usa C. Si no es así el array se copiará en otra zona de memoria y se formateará convenientemente. Cuando llamemos a `ReleaseTipoArrayElements()` la máquina virtual quitará el pinning o liberará la copia del array según proceda.

El JSDK 1.2 se introdujeron las funciones `GetPrimitiveArrayCritical()` y `ReleasePrimitiveArrayCritical()` que se recomienda usar siempre que el hilo no vaya a:

- Llamar a otra funciones JNI
- Meterse en un bucle de sondeo infinito
- Realizar operaciones que duerman al hilo

Si se cumplen estas condiciones la máquina virtual puede desactivar temporalmente la recogida de basura mientras que da al método nativo acceso directo al array, con lo que no es necesario que la máquina virtual use pinning. Esto aumenta las posibilidades de que la máquina virtual devuelva un puntero al array original en vez de hacer una copia.

Por último queda decir que JNI debe garantizar el acceso concurrente al array por varios hilos, con lo que JNI debe mantener un contador de pinnings que ha hecho a un array para que un hilo no quite el pinning a un array cuando otro hilo lo esté usando.

6.4.2 Rendimiento en el acceso a los miembros de un objeto

Obtener un field ID o method ID requiere una búsqueda simbólica en base al nombre del miembro, lo cual es relativamente costoso. Para evitar ejecutar varias veces la búsqueda podemos **cachear** los IDs. Para cachear podemos usar dos técnicas:

1. Cachear en variables estáticas de la función nativa, es decir podemos hacer la función así:

```
JNIEXPORT jobject JNICALL Java_MiClase_miMetodo(JNIEnv *env, jclass
clase)
{
 static jfieldID atributo = NULL;
 if (atributo==NULL)
    atributo = (*env)->GetFieldID(...);
 // Usamos atributo
}
```

2. Cachear al inicializar la clase, es decir en el bloque estático podemos llamar a una función que inicializa todos los atributos y métodos que vayamos a necesitar.

```
public class MiClase
{
 ....
 private native static void inicializaIDs();
 static{
        inicializaIDs();
       }
}
```

Y luego `inicializaIDs()` guarda los IDs en variables globales. De las dos técnicas es recomendable la segunda por varias razones:

1. Los IDs son sólo validos hasta que se descarga la clase. Si el class loader descarga la clase y luego la vuelve a cargar, no se actualizan los IDs de la primera solución, pero en la segunda solución se vuelve a inicializar.

2. Es relativamente más rápida la segunda forma ya que una vez cargados los IDs no tenemos que volver a comprobar el valor de ID con:

```
if (atributo==NULL)
   {
    ..............
   }
```

6.4.3 Diferencias de coste entre métodos nativos y métodos Java

En este apartado vamos a comentar brevemente cuál es el coste de realizar una llamada Java-nativo, o nativo-Java (llamada callback), comparado con realizar una llamada Java-Java. Respecto a una llamada Java-nativo comparada con una llamada Java-Java, la primera es más lenta porque:

1. Como veremos más adelante, la máquina virtual tiene que liberar las referencias locales después de llamar a un método nativo.
2. La máquina virtual tiene que buscar la función en la librería de enlace dinámico.
3. La máquina virtual muchas veces hace llamadas inline cuando un método java llama a otro método java.

Se estima que el coste de la llamada Java-nativo es dos o tres veces el de una llamada Java-Java. También se estima que el coste de una llamada nativo-Java (llamada callback) es dos o tres veces superior al de una llamada Java-Java. El principal coste añadido está en:

1. La obtención de los field ID y method ID
2. En los atributos tenemos que acceder a ellos a través de funciones, mientras que la máquina virtual accede a ellos directamente.

7. Referencias locales y globales

7.1. Qué son las referencias

JNI representa los objetos y los arrays mediante referencias de tipos como `jobject`, `jclass`, `jstring`, o `jarray`. Una característica de estás referencias es que son referencias opacas, es decir, el programador nunca inspecciona la zona a la que apunta la referencia, sino que usa funciones JNI que se encargan de ello. Esta opacidad hace que el programador no se tenga que preocupar de cómo están implementadas realmente estas referencias en cada máquina virtual. En principio esta opacidad puede dar la impresión de que el programador tampoco tiene que saber más sobre las referencias, sin embargo hay una serie de aspectos sobre las referencias que conviene aclarar, y lo vamos a hacer en este apartado.

Es importante diferenciar entre **referencias JNI** y **objetos Java**. Una referencia JNI es una variable de los tipos anteriores (`jobject`, `jclass`, `jstring`, o `jarray`), que apunta a una estructura opaca con información sobre un objeto Java. El objeto Java, es en sí el objeto de la maquina virtual al que accedemos a través de la referencia.

Los tipos de datos fundamentales en Java se pasan por valor, y en JNI también se copian directamente de la máquina virtual al método nativo al que llamamos. Los objetos y arrays en Java se pasan por referencia, y a JNI también se le pasan por referencia, pero al pasar un objeto a JNI no se pasa directamente un puntero al objeto Java, sino que se pasa un puntero a una estructura que es la que controla al objeto Java. A esta estructura se le llama **controlador de objeto**, de los cuales hay una por objeto Java, y es la que se encarga entre otras cosas de llevar la cuenta del número de referencias al objeto. La Figura 5 resume esta idea.

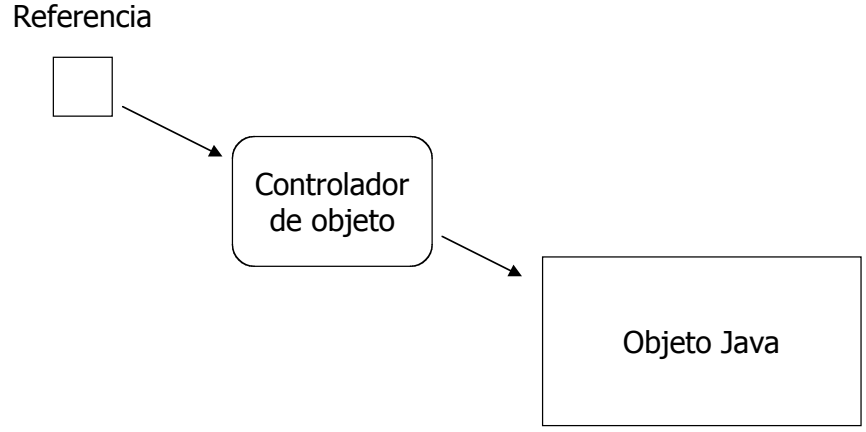

Figura 5: Controlador de objeto en referencias Java

El uso de un controlador de objeto se pensó para que la máquina virtual pueda cambiar la posición de memoria que ocupa el objeto Java sin que se entere la referencia.

7.2. Tipos de referencias

En JNI existen tres tipos de referencias: Referencias locales, referencias globales y referencias globales desligadas. Vamos a ver qué características tienen cada una de ellas.

7.2.1 Referencias locales

Una referencia local es una referencia que es válida sólo durante la llamada a un método nativo. Estas referencias son liberadas por JNI una vez acaba la llamada al método nativo. La mayoría de las funciones de JNI producen referencias locales. Y en general se puede seguir la regla de que si no se especifica lo contrario la función JNI producirá una referencia local.

Las referencias locales no se deben cachear usando técnicas como la explicada en el apartado 6.4.2 , ya que una vez acaba la llamada a la función nativa, Java libera la estructura a la que apunta la referencia. Es decir no debemos de hacer cosas como esta:

```
JNIEXPORT jobject JNICALL Java_MiClase_miMetodo(JNIEnv *env, jobject
obj)
{
 static jclass clase = NULL;
 if (clase==NULL)
    clase = (*env)->FindClassD(...);
 // Usamos clase
}
```

Cuando explicamos que podíamos cachear los field ID que obteníamos de `GetFieldID()`, esto era porque `GetFieldID()` no devuelve una referencia, sino un ID, y los ID sí que son válidos hasta que se descarga la clase, pero el `jclass` es una referencia y no se debe cachear. En breve veremos cómo se debería haber cacheado la referencia. Además las referencias locales son válidas sólo dentro del hilo que las creó, con lo que una referencia creada en un hilo no debe pasarse a otro hilo. En JNI existen dos formas de invalidar una referencia: La primera es dejar que la máquina virtual libere la referencia al acabar la llamada al método nativo, como hemos explicado. La segunda es usar la siguiente función JNI para liberar la referencia:

```
void DeleteLocalRef(JNIEnv* env, jobject lref);
```

Aunque normalmente no liberaremos la referencia, sino que esperaremos al final de la llamada nativa para que JNI la libere, sí que hay casos en los que conviene liberar la referencia local usando estas funciones para evitar un excesivo uso de memoria. Vamos a comentar situaciones en las que se recomiendo usar `DeleteLocalRef()`:

EL primer caso es cuando necesitamos crear un gran número de referencias en una sola función nativa. En este caso podríamos provocar un overflow de la tabla de referencias locales de JNI. Un buen ejemplo es el programa que hicimos en el apartado 5.5.2 que sumaba dos matrices:

```
JNIEXPORT jobjectArray JNICALL Java_Parametros_sumaMatrices
  (JNIEnv *env, jobject obj, jobjectArray arrayA, jobjectArray
arrayB)
{
```

```
 ......
 ......
 for(i=0;i<n_vectores;i++)
    {
     jintArray vectorA = (jintArray)
        (*env)->GetObjectArrayElement(env,arrayA,i);
     jintArray vectorB = (jintArray)
        (*env)->GetObjectArrayElement(env,arrayB,i);
     jintArray vectorC =
        (*env)->NewIntArray(env,longitud_vector);
     .......
     (*env)->DeleteLocalRef(env,vectorA);
     (*env)->DeleteLocalRef(env,vectorB);
     (*env)->DeleteLocalRef(env,vectorC);
    }
 return arrayC;
}
```

Aquí entrábamos en un bucle que creaba una referencia por cada vector de la matriz, y si la matriz tiene muchos vectores podemos agotar las referencias locales.

Un segundo caso es cuando escribimos funciones de utilidad que van a ser llamadas por métodos nativos JNI. En este caso la función de utilidad debe liberar todas las referencias locales asignadas antes de retornar, porque no sabe cuantas veces va a ser llamada en el contexto de una función nativa.

Un tercer caso es cuando el método nativo no retorna en absoluto, sino que entra en un bucle infinito, p.e. un event dispatch loop.

7.2.2 Referencias globales

Las referencias globales son referencias que sobreviven a la terminación de una llamada a un método nativo, de forma que podemos seguir accediendo a ellas una vez que hacemos otra llamada. Para obtener una referencia global, lo hacemos a partir de una referencia local usando:

```
jobject NewGlobalRef(JNIEnv* env, jobject obj);
```

Luego la función anterior la deberíamos haber hecho así:

```
JNIEXPORT jobject JNICALL Java_MiClase_miMetodo(JNIEnv *env, jobject
obj)
{
 static jclass clase = NULL;
 if (clase==NULL)
    {
     clase = (*env)->FindClassD(...);
     clase = (jclass) (*env)->NewGlobalRef(env,clase);
    }
 // Usamos clase
}
```

En las referencias globales la recogida de basura no se ejecuta hasta que liberamos la referencia usando:

```
void DeleteGlobalRef(JNIEnv* env, jobject gref);
```

Todas las funciones Java están diseñadas de forma que aceptan tanto referencias locales como globales indistintamente. Además nuestras funciones nativas pueden retornar indistintamente referencias locales o globales. Pero, ¿cómo están implementadas las referencias locales y globales?.

Cuando llamamos a un método nativo, JNI crea una **tabla de referencias locales** en la que apunta todas las referencias locales que va dando a un método nativo según éste va llamando a funciones JNI, de esta forma cuando acaba la llamada al método nativo, JNI libera todas las referencias que le ha dado. También JNI tiene una **tabla de referencias globales** en la que apunta todas las referencias globales que están creadas, y las va descontando cuando llamamos a `DeleteGlobalRef()`.

7.2.3 Referencias globales desligadas

Las referencias globales desligadas (weak global references) son un nuevo tipo de referencias introducidas en el JSDK 1.2 que al igual que las referencias globales no se destruyen al acabar la llamada al método nativo, con lo que permanecen validas entre diferentes llamadas a métodos nativos, pero a diferencia de las referencias globales, no garantizan que no se ejecutará la recogida de basura sobre el objeto apuntado. Para crear una referencia global desligada se usa la función JNI:

```
jweak NewWeakGlobalRef(JNIEnv* env, jobject obj);
```

Y para liberarla se usa la función:

```
void DeleteWeakGlobalRef(JNIEnv* env, jweak wref);
```

Su principal utilidad es permitirnos mantener cacheadas referencias a objetos en nuestras funciones nativas sin que esto implique que la máquina virtual no pueda liberar el objeto Java al que apuntan si dentro de Java ya no quedan referencias apuntándolo. Obsérvese que ésta es la forma natural de funcionar de una máquina virtual, porque lo que busca es liberar un objeto que nadie está usando. Ahora el problema es que cuando vamos a usar la referencia global desligada tenemos que saber si todavía existe el objeto Java al que apunta la referencia o ha sido liberado por la recogida de basura de la máquina virtual. En el siguiente apartado vamos a comentar cómo hacemos esta comprobación.

Obsérvese que aunque la máquina virtual libere al objeto Java cuando nadie lo necesite, nosotros todavía tenemos que seguir llamando a `DeleteWeakGlobalRef()`, ya que esta función libera la cuenta de referencias en el controlador de objeto a la que apunta la referencia y no el objeto Java, que se libera cuando nadie lo apunta.

7.3. Comparación de referencias

Dadas dos referencias locales, globales, o globales desligadas, siempre podemos comprobar si dos referencias se refieren al mismo objeto Java usando la función:

```
jboolean IsSameObject(JNIEnv* env, jobject ref1, jobject ref2);
```

La función devuelve `JNI_TRUE` o `JNI_FALSE`. Además en JNI existe la regla de que una referencia `null` de Java se representa por `NULL` de C. Con lo que si se cumple

que `mi_obj==NULL` entonces `(*env)->IsSameObject(env,mi_obj,NULL)` también se cumple. Pero no al revés, es decir, si `(env)->IsSameObject(env,mi_obj,NULL)` no tiene porque cumplirse que `(mi_obj==NULL)`.

`IsSameObject()` también es la función que usan las referencias globales desligadas para saber si la referencia todavía apunta a un objeto Java válido, o el objeto Java ha sido liberado por la recogida de basura. Para ello podemos hacer:

```
if ( ! (*env)->IsSameObject(env,mi_obj_desligado,NULL) )
   {
    // La referencia continua siendo válida
   }
```

7.4. Gestión de referencias locales en JSDK 1.2

En el JSDK 1.2 se han introducido funciones de gestión de referencias adiciónales que vamos a comentar.

En la especificación de JNI se dice que la máquina virtual debe asegurar que como mínimo un método nativo debe poder obtener 16 referencias locales. Experimentalmente se ha comprobado que este número de referencias es suficiente para la mayoría de los métodos nativos. Si por alguna razón esto no fuera suficiente, la función nativa puede pedir una garantía de poder obtener más referencias locales usando:

```
jint EnsureLocalCapacity(JNIEnv* env, jint capacity);
```

A la que la pasamos el número mínimo de referencias que queremos que nos garantice la máquina virtual. La función retorna `0` si tiene éxito, o un número menor de `0` si no puede garantizarnos el número de referencias pedidas.

Para saber si alguna función nativa obtiene más de 16 referencias locales (lo cual es peligroso porque podría pasar que el sistema no nos la pudiera dar) podemos usar la opción `-verbose:jni` del comando `java` que nos da un mensaje de advertencia si alguna función crea más de 16 referencias locales durante la ejecución del programa.

Otra posible forma de obtener más de 16 referencias locales es crear **ámbitos anidados de referencias locales**, que consiste en crear un nuevo ámbito en el que podemos crear hasta otras 16 referencias locales usando la función:

```
jint PushLocalFrame(JNIEnv* env, jint capacity);
```

A la función la podemos decir qué cantidad de referencias locales queremos que nos garantice y retorna `0` si tiene éxito o un número negativo si falla. Para destruir el último ámbito creado usamos la función:

```
jobject PopLocalFrame(JNIEnv* env, jobject result);
```

Lo cual implica liberar todas las referencias locales de ese ámbito. De hecho estas son las mismas funciones que usa Java cuando ejecuta un método nativo para crear la tabla de referencias locales y liberarlas al acabar la llamada. La ventaja que aporta el uso de ámbitos es que permiten gestionar el ciclo de vida de un grupo de referencias locales sin tener que preocuparnos por cada referencia local que crea el programa. P.e. si hacemos:

```
JNIEXPORT jobject JNICALL Java_MiClase_miMetodo(JNIEnv *env, jobject
obj)
{
 .......
 .......
 for (int i=0;i<longitud;i++)
     {
      (*env)->PushLocalFrame(env,MAX_REFS);
      .......
      .......
      (*env)->PopLocalFrame(env,NULL);
     }
}
```

Obsérvese que `PopLocalFrame()` recibe como parámetro una referencia en el parámetro `result`, esta referencia la recibe porque a veces es útil que una referencia de un ámbito de referencias locales sobreviva a la terminación de ese ámbito con la llamada a `PopLocalFrame()`. Para ello pasamos la referencia que queremos que sobreviva en `result` y obtenemos en `PopLocalFrame()` otra referencia local en el nuevo ámbito en el que estemos.

```
JNIEXPORT jobject JNICALL Java_MiClase_miMetodo(JNIEnv *env, jobject
obj)
{
 jobject solucion;
 (*env)->PushLocalFrame(env,10);
 .......
 .......
 // La referencia local a la solucion
 // la obtenemos en este ámbito
 solucion = ......
 .......
 .......
 solucion = (*env)->PopLocalFrame(env,solucion);
 // La referencia local a la socucion está ahora
 // en el ámbito superior
 return solución;
}
```

Estas funciones también las podemos usar si vamos a llamar a una función de utilidad que consume muchas referencias locales.

8. Excepciones

En este apartado vamos a ver cómo funciona el manejo de excepciones en JNI. Antes de nada debe quedar claro que las excepciones sólo se producirán cuando llamemos a funciones de JNI o a métodos Java desde un método nativo (llamadas callback). Cuando llamemos a funciones del SO (APIs) o la librerías, debemos de seguir el mecanismo de gestión de errores del SO o la librería que estemos usando, que normalmente suelen aprovechar el retorno de la función para devolver un código de error.

8.1. Capturar excepciones en un método nativo.

Cuando se produce una excepción (bien sea en un método JNI o en un método callback de Java) no se interrumpe el flujo normal de ejecución del programa (como pasa en Java) sino que en el hilo se activa un flag indicando que se ha lanzado la excepción. Existe un único flag por cada hilo y éste está almacenado en la estructura `env`, con lo que el hecho de que se produzca una excepción en un hilo no afecta a los demás hilos. Podemos comprobar si en nuestro hilo se ha producido una excepción llamando a:

```
jboolean ExceptionCheck(JNIEnv* env);
```

O alternativamente podemos usar:

```
jthrowable ExceptionOccurred(JNIEnv* env);
```

Esta última función devuelve una referencia a la excepción ocurrida o `NULL` si no ha habido excepción. Auque parezca tedioso es necesario que una función nativa bien construida compruebe si se ha producido una excepción cada vez de llama a una función JNI que puede producir una excepción o a un método callback de Java. Para simplificar este trabajo existe la regla de que si una función JNI que debería devolver una referencia o identificador devuelve `NULL`, entonces obligatoriamente se debe haber producido una excepción, con lo que hacer:

```
jfieldID fieldID = (*env)->GetFieldID(env,clase,"atributo","I");
if (fieldID==NULL)
   {
    // Se ha producido una excepcion y la tratamos
   }
```

Es equivalente a hacer:

```
jfieldID fieldID = (*env)->GetFieldID(env,clase,"atributo","I");
if ((*env)->ExceptionCheck())
   {
    // Se ha producido una excepcion y la tratamos
   }
```

También hay funciones JNI que no devuelven un código especial cuando se produce una excepción como p.e. `CallVoidMethod()` que aunque no retorna nada activa el flag de excepción en el hilo si se produce una excepción en la llamada al método

callback Java. En este caso también debemos comprobar la excepción con `ExceptionCheck()`

```
(*env)->CallVoidMethod(...);
if ((*env)->ExceptionCheck())
   {
    // Se ha producido una excepcion y la tratamos
   }
```

En cualquier caso, si nuestro método nativo no se molesta en comprobar las posibles excepciones de cada llamada pueden producirse resultados inesperados al seguir llamando a más funciones de JNI.

En realidad, cuando se produce una excepción hay una pequeña lista de funciones JNI que son las únicas que se pueden ejecutar de forma segura antes de tratar la excepción que es la siguiente. Estas funciones nos permiten tratar la excepción o liberar recursos antes de retornar a Java:

```
ExceptionOcurred()          ReleaseStringChars()
ExceptionDescribe()         ReleaseStringUTFChars()
ExceptionClear()            ReleaseStringCritical
ExceptionCheck()            ReleaseTipoArrayElements()
DeleteLocalRef()            ReleasePrimitiveArrayCritical()
DeleteGlobalRef()           MonitorExit()
DeleteWeakFlobalRef()
```

Una vez que se produce una excepción, ésta permanece activa hasta que una de estas tres circunstancias se produzcan:

1. Desactivamos el flag de la excepción con:

```
void ExceptionClear(JNIEnv* env);
```

Esta función hace lo mismo que el bloque `catch` de Java, que también desactiva la excepción una vez tratada.

2. Retornamos del método nativo con lo que la excepción se lanza en el programa Java.

3. Cuando se produce una excepción, también podemos imprimirla usando:

```
void ExceptionDescribe(JNIEnv* env);
```

Que imprime una traza de la pila de llamadas a funciones en `System.err`, al igual que hace en método Java:

```
void <Throwable> printStackTrace()
```

Un efecto lateral de llamar a `ExceptionDescribe()` es que desactiva el flag de la excepción pendiente.

8.2. Lanzar excepciones desde un método nativo

Una función nativa puede lanzar una excepción usando la función JNI:

```
jint ThrowNew(JNIEnv* env, jclass class, const char* message);
```

Lo cual activa el flag de la excepción y deja la excepción pendiente hasta que la tratemos. La función retorna `0` si la excepción se lanza o un valor distinto de `0` si no se pudo lanzar la excepción (p.e. porque ya había otra lanzada o porque la clase no deriva de `Throwable`)

P.e. para lanzar una `IllegalArgumentException` haríamos:

```
jclass clase = (*env)FindClass(env,
               "java/lang/IllegalArgumentException");
if (clase==NULL)
   return NULL; // NoClassDefFoundError lanzada
(*env)->ThrowNew(env,clase,"Excepcion lanzada desde C");
```

Alternativamente si disponemos ya de una referencia `jthrowable` a una excepción ya creada (p.e. que la hallamos capturado y la queramos volver a lanzar) podemos usar:

```
jint Throw(JNIEnv* jthrowable);
```

8.3. Excepciones asíncronas

Un hilo puede provocar una excepción en otro hilo llamando al método:

```
void <Thread> interrupt()
```

Cuando se produce la excepción el otro hilo, la ejecución del hilo no se interrumpe hasta que:

1. El hilo dentro de un método nativo llama a una función JNI que produce excepciones y el hilo comprueba la excepción con `ExceptionCheck()`
2. El hilo llama a un método Java que produce excepciones.

8.4. Ejemplo

Como ejemplo en el Listado 20 hemos hecho un método nativo que llama a al método Java del Listado 19 llamado `divideJava()`, el cual produce una `ArithmeticException`, captura la `ArithmeticException` y lanza una `IllegalArgumentException`.

```
public class Excepciones
{
 private static int divideJava(int dividendo, int divisor)
         throws ArithmeticException
```

```
        {
         return dividendo/divisor;
        }
 private static native int divideC(int a, int b)
                       throws IllegalArgumentException;
 public static void main(String[] args)
        {
         try{
             System.out.println("Intentando dividir 5/0");
             int cociente = divideC(5,0);
             System.out.println("Division correcta:"+cociente);
            }
         catch(Exception ex){
             ex.printStackTrace();
            }
        }
 static{
        System.loadLibrary("Excepciones");
       }
}
```

Listado 19: Clase Java con método que lanza una excepción

```
JNIEXPORT jint JNICALL Java_Excepciones_divideC(JNIEnv * env, jclass
clase, jint a, jint b)
{
 jint sol;
 jmethodID methodID = (*env)->GetStaticMethodID(env,clase
                                    ,"divideJava","(II)I");
 if (methodID==NULL)
    return NULL; // Excepcion lanzada
 sol = (*env)->CallStaticIntMethod(env,clase,methodID,a,b);
 if ((*env)->ExceptionCheck(env))
    {
     jclass clase_excepcion;
     (*env)->ExceptionDescribe(env);
     clase_excepcion = (*env)->FindClass(env
                 ,"java/lang/IllegalArgumentException");
     if (clase_excepcion==NULL)
         return NULL; // Excepcion lanzada
     (*env)->ThrowNew(env,clase_excepcion
                         ,"El divisor no es valido");
     return 0;
    }
 return sol;
}
```

Listado 20: Función nativa que captura excepción Java

Y al ejecutar el programa obtenemos:

```
$ java Excepciones
Intentando dividir 5/0
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Excepciones.divideJava(Excepciones.java:5)
        at Excepciones.divideC(Native Method)
        at Excepciones.main(Excepciones.java:12)
java.lang.IllegalArgumentException: El divisor no es valido
        at Excepciones.divideC(Native Method)
        at Excepciones.main(Excepciones.java:12)
```

8.5. Consideraciones de rendimiento

Las funciones JNI no comprueban que los parámetros que las pasemos sean correctos, con lo que pasar parámetros incorrectos a una función puede dar lugar a errores inesperados. Esta decisión se tomó con el fin de mejorar el rendimiento de JNI. Aunque no es necesario que una máquina virtual comprueba los parámetros, se anima a los fabricantes de máquinas virtuales a comprobar los errores de programación más comunes.

Además a partir del JSDK 1.2 existe una opción no estándar `-Xcheck:jni` que indica a la máquina virtual que queremos comprobar los parámetros que pasamos a las funciones de JNI. Aunque esto va en contra del rendimiento, es una opción que se usa sólo durante la fase de pruebas, y ayuda a corregir errores comunes del programador, como p.e. confundir un `jclass` con un `jobject`.

Parte II

Programación avanzada con JNI

Sinopsis:

En esta segunda parte se empieza tratando el tema de la programación multihilo cuando se hace desde librerías nativas. Después se tratará el tema de la instanciación y uso de máquinas virtuales desde aplicaciones nativas. También comentaremos cómo hace Java para cargar las librerías de enlace dinámico, y cómo conseguir que Java acceda a librerías nativas no diseñadas para ser llamadas desde Java. Acabaremos viendo cómo acceder a la información de introspección desde JNI, qué diferencias hay a la hora de acceder a JNI desde C++, y cómo manejar distintos juegos de caracteres desde JNI.

1. Programación multihilo con JNI

La máquina virtual Java soporta el tener múltiples hilos ejecutando en el mismo espacio de memoria de un proceso. Esta concurrencia introduce un grado de complejidad ya que varios hilos pueden intentar acceder a la vez a un mismo recurso. En este apartado vamos a comentar aspectos propios de JNI en la programación multihilo, y supondremos que el lector ya conoce la programación multihilo en Java. Si el lector no tiene conocimientos sobre este tema desde el punto de vista de Java deberá remitirse primero a un libro que trate este tema.

1.1. Restricciones para JNI

Hay dos restricciones que debemos tener en cuenta cuando escribamos un método nativo que va a ser ejecutado en un entorno multihilo. Estas restricciones se deben tener en cuenta para evitar problemas debidos a la ejecución simultanea de ese método por varios hilos, que muchas veces se conocen como race conditions:

1. El puntero `JNIEnv` es válido solamente en el hilo asociado a él. Nunca debemos pasar este puntero a otro hilo, para ello debemos evitar el típico error de cachear el puntero `JNIEnv`. Podemos estar seguros de que si la máquina virtual llama varias veces a un método nativo usando el mismo hilo, siempre pasará el mismo puntero `JNIEnv`, pero si lo llama desde distintos hilos, pasará distintas instancias del puntero `JNIEnv`.
2. Las referencias locales son válidas sólo dentro del hilo que las creó. Si queremos pasar una referencia a otro hilo, deberemos convertir la referencia primero en una referencia global.

1.2. La interfaz `JNIEnv`

Como ya sabemos, los métodos nativos acceden a los servicios de JNI a través del puntero `JNIEnv`. También ya hemos explicado que este puntero es un puntero a una estructura donde hay información propia de cada hilo. Esta información es opaca al programador. Como muestra la Figura 6, el primer campo de estas informaciones es un puntero a una tabla de punteros a funciones.

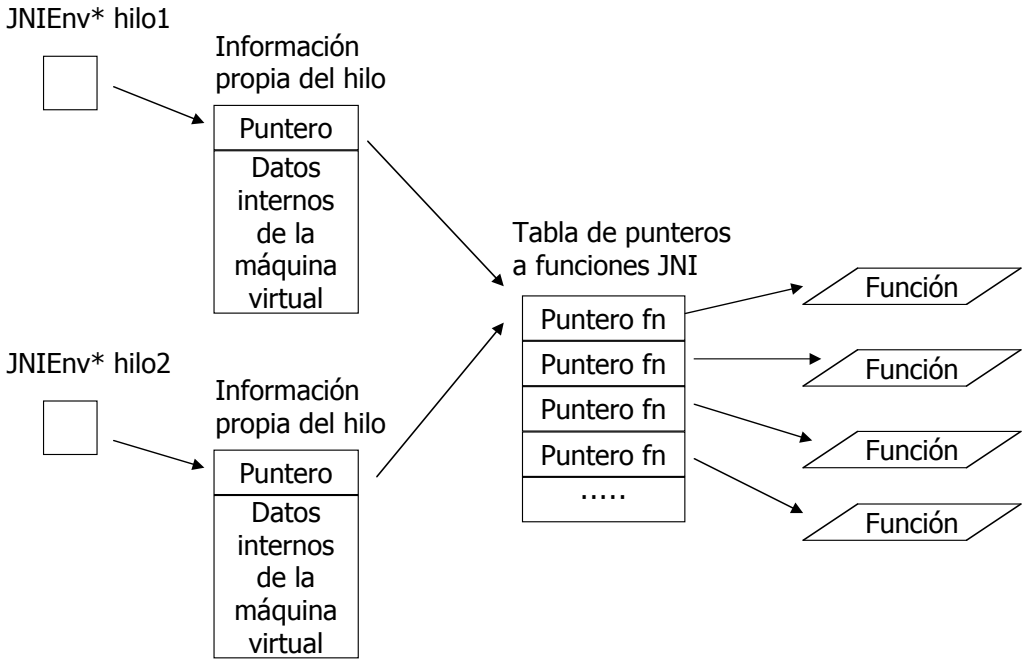

Figura 6: Interfaz `JNIEnv` de varios hilos

Esta organización es equivalente a la de los interfaces COM de Microsoft, y de hecho fue pensada inicialmente para mantener compatibilidad con estos, y de hecho las tres primeras entradas de la tabla de punteros a funciones están reservadas para compatibilidad con COM, y de momento siempre valen `NULL`. Por eso al puntero `JNIEnv` también se le llama la interfaz `JNIEnv`, porque actúa como una interfaz COM. Obsérvese que aunque la estructura con información de cada hilo es distinta, la tabla de punteros a funciones es compartida por todos los hilos. Como hemos dicho, la máquina virtual nos garantiza que se pasara el mismo puntero a métodos nativos llamados por el mismo hilo, y que se pasaran punteros distintos a métodos nativos llamados por distintos hilos. Con lo que los métodos nativos pueden usar el puntero `JNIEnv` como un ID que permanece único durante la vida del hilo. Las razones por la que se decidió usar las interfaces `JNIEnv` son básicamente dos:

1. La estructura del puntero `JNIEnv` tiene información propia del hilo, ya que en muchas plataformas no existe una forma eficiente de almacenar información propia de cada hilo.
2. Como la tabla de punteros a funciones se pasa siempre a los métodos nativos, estos no tienen que enlazar una llamada a un método exportado en alguna librería de enlace dinámico, lo cual es crucial para que un mismo método nativo pueda funcionar el varias implementaciones de la máquina virtual un mismo host

environment. Si los métodos nativos llamaran a funciones exportadas por una máquina virtual, que están publicados en ficheros de librería de enlace dinámico distintas, al cambiar la maquina virtual tendría que usar una librería de enlace dinámico distinta, sin embargo, si usa punteros a funciones, simplemente ejecuta el método que esté cargado en la dirección de memoria que le diga la tabla de punteros a funciones, pero al método nativo le da lo mismo que máquina virtual creó la tabla de punteros a funciones.

Otra ventaja importante de usar punteros a funciones es que la máquina virtual puede proporcionar dos juegos de funciones JNI en dos tablas de punteros a funciones distintas. Una tabla con funciones de depuración y otra con funciones de explotación. P.e. es muy típico a partir del JSDK 1.2 que la máquina virtual soporte la opción `-Xckeck:jni` que proporciona una tabla de punteros a funciones con funciones que comprueban el tipo y valor de los parámetros para detectar posibles errores en los parámetros pasados por el programador.

1.3. Monitores

Los monitores son el principal mecanismo de sincronización entre hilos que tiene Java. Cada objeto Java tiene asociado un monitor, de forma que como máximo un hilo puede poseer el monitor. En el lenguaje Java existen dos formas de apoderarse de un monitor:

1. Entrar en una sección crítica indicada por un bloque `synchronized`:

```
synchronized (obj) {
  ··· // Sección crítica
 }
```

2. Llamar a un método síncrono, marcado por el modificador `synchronized`

```
public snchronized void miMetodoSincrono()
       {
        ··· // Sección crítica
       }
```

Cuando un hilo entra en una sección crítica (se apodera del monitor), todos los demás tienen que esperar hasta que éste acabe. Desde un método nativo también podemos apoderarnos de un monitor, para lo cual tenemos las funciones JNI:

```
jint MonitorEnter(JNIEnv* env, jobject obj);
jint MonitorExit(JNIEnv* env, jobject obj);
```

Ambas funciones retornan `0` si tienen éxito o un número negativo si hay error, en cuyo caso se lanza una excepción con el error producido, que puede ser una excepción del tipo `OutOfMemoryError` si el sistema se queda sin memoria o una excepción del tipo `IllegalStateException` si hemos pedido liberar un monitor que no poseemos. En caso de que el monitor esté poseído por otro hilo, nuestro hilo queda suspendido hasta que el otro hilo libere el monitor. Luego la forma de llamar a estas funciones debe comprobar siempre el retorno de la llamada así:

```
if ((*env)->MonitorEnter(env,obj) != JNI_OK)
   ... // Tratamos el error

// Sección crítica
.................

if ((*env)->MonitorExit(env,obj) != JNI_OK)
   ··· // Tratamos el error
```

Hay que tener en cuenta que si durante la sección crítica se produce una excepción, debemos de liberar el monitor llamando a `MonitorExit()` antes de retornar del método nativo, o dejaremos el monitor bloqueado para siempre.

```
if ((*env)->MonitorEnter(env,obj) != JNI_OK)
   ... // Tratamos el error

// Sección crítica
.................
if ((*env)->ExceptionOccurred(env))
   {
    (*env)->MonitorExit(env,obj);
    return;
   }

if ((*env)->MonitorExit(env,obj) != JNI_OK)
   ··· // Tratamos el error
```

Obsérvese que si aquí fallase la llamada a `MonitorExit()` el monitor quedaría bloqueado para siempre. Por razones como esta, en general se recomienda usar monitores en Java y no en el código nativo, para ello podemos declarar al método nativo como `synchronized`. Pero hay ocasiones en la que esto no es posible, como p.e. cuando el método nativo realiza una operación larga, y sólo una pequeña parte del método nativo necesita ejecutarse en una sección crítica.

1.4. wait() notify() y notifyAll()

La API de Java tiene también los métodos:

```
void <Object> wait()
void <Object> notify()
void <Object> notifyAll()
```

que son métodos que sólo se pueden ejecutar dentro de una sección crítica (cuando el hilo posee el monitor). En JNI no existen funciones equivalentes a estos métodos, sino que si quisiéramos llamarlos tendríamos que hacerlo con una llamada callback a un método Java. La razón que alegan los diseñadores de JNI para no incluir funciones JNI equivalentes es que las llamadas a estos métodos se hacen en situaciones donde el rendimiento no es tan crítico, como ocurre en las situaciones en las que se entra y sale de una sección crítica sin tener que esperar.

1.5. El modelo multihilo de la máquina virtual y del host environment

La máquina virtual es una máquina virtual multihilo, pensada de forma que en el mismo espacio de memoria se pueden tener varios hilos en ejecución. Pero hay sistemas operativos que no disponen de un modelo multihilo. P.e. los "green thread" de las antiguas versiones de Solaris o Mac OS Classic son sistemas operativos que no soportan el modelo multihilo, sino un modelo multiproceso. Para solucionar este problema la máquina virtual utiliza técnicas que emulan que la máquina virtual se está ejecutando en un sistema multihilo. En otros casos el sistema operativo sí soporta el modelo multihilo donde un proceso puede tener en ejecución varios hilos. P.e. Mac OS X, Win 32, OS/2 y BeOS son sistemas operativos que soportan completamente el modelo multihilo.

Cuando el modelo multihilo del host environment coincide con el modelo multihilo de la máquina virtual es posible usar llamadas a primitivas del SO para realizar operaciones multihilo de la máquina virtual. P.e. en estos sistemas podemos crear un hilo o bien instanciando un objeto de tipo `java.lang.Thread` y llamando a `start()` sobre el objeto, o bien llamando a funciones nativas del sistema operativo.

La creación nativa de hilos dependen de la plataforma en la que estemos: En Mac OS X crearemos un objeto `NSThread` si estamos trabajando con Objective-C o crearemos un `pthread` de POSIX si estamos trabajando con C. En Solaris 8.0 podemos llamar a la función `thr_create()` clásica o bien usar pthreads con `pthread_create()` En Win32 usamos la función `CreateThread()`

Análogamente las llamadas a `MonitorEnter()` y `MonitorExit()` de JNI son equivalentes a llamar a `NSLock::lock()` y `NSLook::unlock()` en Mac OS X o a `mutex_lock()` en Solaris 8.0. Para una mejor descripción del modelo multihilo de Mac OS X consulte [OSXTHREADING].

Por otro lado si el modelo multihilo del host environment no coincide con el modelo multihilo de la máquina virtual, estas operaciones no son posibles, ya que la máquina virtual se basa en una librería de apoyo que crearon los implementadores de la máquina virtual. En este caso podríamos enlazar llamadas a estas librerías creados por los diseñadores de la máquina virtual, pero esta sincronización se puede volver más complicada de lo que puede parecer a simple vista, ya que muchas funciones de librería de C (p.e. entrada/salida o asignación de memoria) realizan operaciones de sincronización que no son controladas desde la librería Java, con lo que implementar métodos nativos en estos host environment se vuelve muy complicado. Afortunadamente hoy en día todos los sistemas operativos modernos utilizan un modelo multihilo similar al de Java.

2. El Invocation Interface

En este capitulo vamos a ver cómo se hace para incrustar una máquina virtual en una aplicación nativa. Esto es especialmente útil para poder construir aplicaciones como:

- Una aplicación nativa que ejecute un programa Java. P.e. el comando `java`
- Un browser que ejecute applets
- Un servidor web que ejecute servlets

La máquina virtual que queremos usar desde una aplicación nativa se distribuye como una librería de enlace dinámico. La aplicación lo que tiene que hacer para cargar una instancia de la máquina virtual Java es enlazarse con está librería y llamar a unas funciones de esta librería que es a lo que se llama la **invocation interface**.

2.1. Creación de una máquina virtual

Para que una aplicación nativa cree una instancia de la máquina virtual debe usar la función:

```
jint JNI_CreateJavaVM(JavaVM** pvm, void** penv, void* vm_args);
```

La función devuelve `0` si tiene éxito o un número negativo si no.

El primer parámetro `pvm` es un puntero a un `JavaVM*` donde se nos deposita un puntero a la máquina virtual. Este puntero lo usaremos luego cuando queramos referirnos a la instancia de la máquina virtual. Aunque actualmente sólo puede haber una instancia de máquina virtual por proceso, en el futuro se pretende que podamos instanciar varias máquinas virtuales en el mismo proceso. Una vez que se instancia la máquina virtual, el hilo que llamó a la función se convierte en el hilo main de la máquina virtual.

El parámetro `penv` es un puntero a un `JNIEnv*` donde se nos deposita el puntero `JNIenv` con información del hilo main. Por último `vm_args` lleva argumentos de creación de la máquina virtual. Estos argumentos varían dependiendo de si estamos en el JSDK 1.1 o en el JSDK 1.2 o posterior. Vamos a comentar qué llevan en cada caso:

En el JDSK 1.1 `vm_args` es un puntero a una estructura de esta forma que se muestra en el Listado 21.

```
typedef struct JDK1_1InitArgs {
    jint version;

    char **properties;
    jint checkSource;
    jint nativeStackSize;
    jint javaStackSize;
    jint minHeapSize;
    jint maxHeapSize;
```

```
    jint verifyMode;
    char *classpath;

    jint (JNICALL *vfprintf)(FILE *fp
         , const char *format, va_list args);
    void (JNICALL *exit)(jint code);
    void (JNICALL *abort)(void);

    jint enableClassGC;
    jint enableVerboseGC;
    jint disableAsyncGC;
    jint verbose;
    jboolean debugging;
    jint debugPort;
} JDK1_1InitArgs;
```

Listado 21: Parámetros en JSDK 1.1

Donde el campo `version` siempre debe llevar el valor `JNI_VERSION_1_1` los demás campos de la estructura se describen en la Tabla 6.

Campo	Descripción
`properies`	Array de propiedades del sistema
`checkSource`	Comprobar si los ficheros `.java` son más recientes que sus correspondientes `.class`
`nativeStackSize`	Tamaño máximo de la pila del hilo main
`javaStackSize`	Tamaño máximo de la pila a usar por cada `java.lang.Thread` de la máquina virtual
`minHeapSize`	Tamaño inicial del heap usado por la máquina virtual
`maxHeapSize`	Tamaño máximo de heap que puede usar la máquina virtual
`verifyMode`	Qué bytescodes debe chequear la máquina virtual: `0`-ninguno, `1`- Los cargados remotamente (p.e. applets), `2`-Todos los bytecodes.
`classpath`	`CLASSPATH` del sistema de ficheros local
`vfprintf`	Si no vale `NULL` debe ser una función de tipo `printf()` de C a la que queremos que la máquina virtual redirija todos los mensajes. Útil para el desarrollo de IDEs.
`exit`	Una función llamada por la máquina virtual justo antes de acabar, lo cual es útil para liberar recursos al final.
`abort`	Función llamada por la máquina virtual si abortamos la ejecución con Ctrl+C
`enableClassGC`	Habilitar la eliminación de clases durante la recogida de basura
`enableVerboseGC`	Mostrar mensajes cuando se realicen recogidas de basura
`disableAsyncGC`	Permitir o no la recogida de basura asíncrona
`reserved0`	Campos reservados para ampliaciones futuras.
`reserved1`	
`reserved2`	

Tabla 6: Campos de JSDK1_`1InitArgs`

Si no queremos rellenarnos esta estructura tan grande a mano podemos usar la función:

```
void JNI_GetDefaultJavaVMInitArgs(void* vm_args);
```

que nos rellena la estructura con valores por defecto. Antes de llamar a la función debemos de haber puesto en `version` el valor `JNI_VERSION_1_1`. La función anterior sólo sirve para rellenar la estructura de JSDK 1.1, la estructura del JSDK 1.2 no se puede rellenar apoyándonos en esta función.

En JSDK 1.2 se decidió cambiar la forma de pasar los argumentos en `vm_args` para conseguir poder pasar cualquier tipo de información de inicialización como pares `clave=valor`. Con esto se consigue que además de las opciones estándar definidas por JavaSoft, en unas máquinas virtuales se puedan pasar opciones adicionales propias de esa implementación de máquina virtual. Estas últimas opciones son las que empiezan por `-X` cuando ejecutamos una aplicación con el comando `java`. P.e. la máquina virtual de Sun tiene las opciones `-Xms` y `-Xmx` para indicar el tamaño inicial y máximo del heap. Para pasar estas opciones, en `vm_args` pasamos un puntero a una estructura `JavaVMInitArgs` cuya forma se muestra en el Listado 22.

```
typedef struct JavaVMInitArgs {
    jint version;
    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
} JavaVMInitArgs;
```

Listado 22: Estructura `JavaVMInitArgs`

Ahora el campo `version` debe tener el valor `JNI_VERSION_1_2`, el campo `nOptions` indica el número de opciones que vamos a pasar en `options` y el campo `ignoreUnrecognized` si vale `JNI_TRUE` indica que se pueden ignorar las opciones no estándar (las que empiecen por "`-X`" o por "`_`") si la máquina virtual que estamos usando no entiende alguna de ellas. Las opciones estándar las debe de entender todas o producir un error si no las entiende. Si vale `JNI_FALSE` produce un error cuando encuentra la primera opción que no entiende y `JNI_CreateJavaVM()` retorna `JNI_ERR`. El campo `options` es un puntero a un array de elementos del tipo `JavaVMOption` que se muestra en el Listado 23. El tamaño de este array nos lo da `nOptions`.

```
typedef struct JavaVMOption {
    char *optionString;
    void *extraInfo;
} JavaVMOption;
```

Listado 23: Estructura `JavaVMOption`

Las opciones estándar que se pueden poner en `optionString` son las que se muestran en la Tabla 7.

optionString	Descripción
`-D<name>=<value>`	Fija una propiedad del sistema
`-verbose`	Habilita la salida verbose donde da mensajes de lo que está haciendo.

	A esta opción se la puede preceder por dos puntos y el tipo de mensajes que nos interesa. P.e.: `-verbose:gc,class` indica que queremos información sobre la recogida de basura y las clases que se van cargando. Las posibles opciones que podemos poner aquí son `gc,class,jni` las demás opciones no son estándar y deben empezar por `x`
`vfprintf`	`extrainfo` debe ser un puntero a la función
`exit`	`extrainfo` debe ser un puntero a la función
`abort`	`extrainfo` debe ser un puntero a la función

Tabla 7: Posibles valores de `optionString`

En el JSDK 1.2 todavía se puede pasar una estructura al estilo del JSDK 1.1 que la acepta por compatibilidad, pero no se recomienda. A nosotros más adelante nos va a ser útil para poder ejecutar la máquina virtual como se hacía en el JSDK 1.1

2.2. Enlazar una aplicación nativa con una máquina virtual Java

Nuestra aplicación nativa tiene que enlazar una llamada con la librería de enlace dinámico donde viene implementada la máquina virtual Java. El cómo realizamos este enlace va a depender de si nuestra aplicación nativa va a usar siempre la misma máquina virtual. P.e. la máquina virtual de Apple para Mac OS X la tenemos en `/System/Library/Frameworks/JavaVM.framework/JavaVM` , en Linux y Solaris la tenemos en `libjava.so` en la versión 1.1 o en `libjvm.so` en la versión 1.2 y posteriores, en Windows tenemos la máquina virtual de Sun en `javai.dll` en la versión 1.1 o `jvm.dll` en la versión 1.2 y posteriores.

En los ejemplos que hemos hecho hasta ahora incluíamos la librería de enlace dinámico en la llamada al compilador, pero de hecho no era necesaria, gracias a que el puntero `JNIEnv` nos permitía ser independientes de la librería de enlace dinámico donde esté implementada la máquina virtual. Recuérdese que este puntero apuntaba a una tabla de punteros a funciones que ya había cargado la maquina virtual, y nosotros sólo indireccionabamos el puntero para ejecutar sus funciones. Animamos al lector a que pruebe a enlazar los ejemplos anteriores sin incluir la librería en la llamada al compilador. Ahora, sin embargo, vamos a tener que ejecutar `JNI_CreateJavaVM()`, que es una función exportada en la librería de enlace dinámico de la máquina virtual, luego ahora es necesario proporcionar esta librería al enlazador.

2.2.1 Enlazar con una máquina virtual conocida

Si nuestra aplicación nativa va a usar sólo una determinada máquina virtual podemos enlazar la aplicación nativa con las funciones exportadas por la librería donde está implementada la máquina virtual.

Para enlazar nuestra aplicación en Mac OS X podríamos hacer:

```
$ gcc ejecuta.c -I/System/Library/Frameworks
/JavaVM.framework/Headers -framework JavaVM -o ejecuta
 -lobjc
```

`-framework JavaVM` incluye la librería de enlace dinámico de Java para Mac OS X.
`-lobjc` Es necesario porque `JavaVM` hace llamadas a esta librería

Para enlazarla Linux usaríamos:

```
$ gcc ejecuta.c -L/usr/lib/java/jre/lib/i386/client -ljvm -o ejecuta
```

`-L` indica el directorio donde están las librerías de enlace dinámico de Java
`-ljvm` indica que usemos las funciones de `libjvm.so` que es donde esta la función `JMI_CreateJavaVM()`

En Solaris usaríamos:

```
$ cc ejecuta.c –I/java/include/ -L/java/lib –lthread –ljvm -o ejecuta
```

`-I` indica el directorio donde esta `jni.h`
`-L` indica el directorio donde están las librerías de enlace dinámico de Java
`-lthread` indica que usemos la implementación de máquina virtual con soporte nativo para hilos
`-ljvm` indica que usemos las funciones de `libjvm.so` que es donde esta la función `JMI_CreateJavaVM()`

Y en Windows haríamos:

```
> cl ejecuta.c /I C:\jdk\include /I C:\jdk\include\win32 -MD /link
C:\jdk\lib\jvm.lib
```

`/MD` Asegura que la aplicación nativa se enlaza con la librería C multihilo de Win32. La cual se usa tanto en el JSDK 1.1 como en el JSDK 1.2
`/link C:\JDK\lib\jvm.lib` se pone porque a Windows no le debemos pasar directamente las librerías de enlace dinámico a usar (`.dll`), sino que debemos pasarle un fichero `.lib` con los símbolos de la correspondiente `.dll`. En nuestro caso en vez de pasarle `jvm.dll` le pasamos `jvm.lib` usando esta opción. El fichero `jvm.lib` se debe distribuir junto con la máquina virtual para poder enlazar las llamadas con la `dll`. Sun distribuye el fichero `jvm.dll` en el directorio `%JAVA_HOME%\lib`

2.2.2 Enlazar con una máquina virtual desconocida

Si vamos a hacer un programa nativo que tiene que ejecutar una máquina virtual sin saber de antemano dónde está esa máquina virtual instalada y de qué fabricante es, o bien que debe ejecutar con máquinas virtuales de diferentes fabricantes, no podemos dejar enlazada la llamada a una determinada librería de enlace dinámico. Los sistemas operativos tienen APIs que permiten cargar el fichero de una librería de enlace dinámico en tiempo de ejecución y después pedir un puntero a cualquiera de

las funciones de librería que se han cargado. De nuevo, las APIs a usar varían de SO en SO.

En Mac OS X para cargar una clase que tengamos en una librería desde Objective-C tenemos la clase `NSBundle`, que representa un bundle, del cual luego podemos cargar el código ejecutable de ese bundle usando:

```
- (BOOL)load
```

y después podemos pedir una clase usando:

```
- (Class)classNamed:(NSString *)className
```

P.e. el programa del Listado 24 carga el ejecutable de un bundle, y después carga la clase `FaxWatcher`.

```
- (void)loadBundle:(id)sender
{
    Class exampleClass;
    id newInstance;
    NSString *str =
       @"/me/Projects/BundleExample/BundleExample.bundle";
    NSBundle *bundleToLoad =
         [NSBundle bundleWithPath:str];
    if (exampleClass =
          [bundleToLoad classNamed:@"FaxWatcher"]) {
        newInstance = [[exampleClass alloc] init];
    // [newInstance doSomething];
    }
}
```

Listado 24: Programa que carga una clase Java

Si estamos usando C en vez de Objective-C podríamos usar las funciones POSIX heredadas de FreeBSD, y que también tiene Solaris:

```
dlopen(char *path, int mode)
```

que carga una librería de enlace dinámico y

```
void *dlsym(void *handle, char *symbol)
```

que obtiene un puntero a una función de la librería.

También tenemos una forma de obtener un puntero a función de librería, usando la API C de Carbon, en concreto las función:

```
void *CFBundleGetFunctionPointerForName (
    CFBundleRef bundle,
    CFStringRef functionName
);
```

En Windows tenemos las APIs:

```
HINSTANCE LoadLibrary(LPCTSTR lpLibFileName);
```

Que nos carga en memoria la librería indicada en `lpLibFileName`. La librería retorna un handle a la librería cargada. Después tendremos que pedir un puntero a la función `JNI_CreateJavaVM()` usando la API:

```
FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName);
```

A la que la pasamos el handle que nos devolvió la función anterior y el nombre de la función que queremos ejecutar. Luego para ejecutar desde Mac OS X la función `JNI_CreateJavaVM()` de una librería que nos pasan como parámetro, usando las APIs `dlopen()` y `dlsym()` deberíamos hacer una función como muestra el Listado 25. Y en Windows esta misma función se implementaría como muestra el Listado 26.

```
typedef jint (JNICALL *FN)(JavaVM **pvm, void **penv, void
*args);

jint EjecutaCreateJavaVM(char* vmlibpath
    ,JavaVM** pjvm,JNIEnv** penv,JavaVMInitArgs* pvm_args)
{
 FN fn;
 void* libVM = dlopen(vmlibpath, RTLD_LAZY);
 if (libVM==NULL)
    {
     fprintf(stderr,"Error al cargar la libreria\n");
     return -1;
    }
 fn = (FN)dlsym(libVM,"JNI_CreateJavaVM");
 if (fn==NULL)
    {
     fprintf(stderr
       ,"Error al obtener un puntero a JNI_CreateJavaVM");
     return -1;
    }
  return fn(pjvm,(void**)penv,pvm_args);
}
```

Listado 25: Función OS X que instancia una máquina virtual a partir del path de librería

```
typedef jint (JNICALL *FN)(JavaVM **pvm, void **penv, void
*args);

jint EjecutaCreateJavaVM(LPCTSTR lpPathLibreria
    ,JavaVM** pjvm,JNIEnv** penv,JavaVMInitArgs* pvm_args)
{
 FN fn;
 HINSTANCE hVM = LoadLibrary(lpPathLibreria);
 if (hVM==0)
    {
     fprintf(stderr,"Error al cargar la libreria\n");
     return -1;
    }
 fn = (FN)GetProcAddress(hVM,"JNI_CreateJavaVM");
 if (fn==NULL)
    {
     fprintf(stderr
```

```
       ,"Error al obtener un puntero a JNI_CreateJavaVM");
     return -1;
    }

  return fn(pjvm,(void**)penv,pvm_args);
}
```

Listado 26: Función Windows que instancia una máquina virtual a partir del path de librería

2.3. Ejemplo: Aplicación nativa que ejecuta un programa Java en una máquina virtual incrustada

En el Listado 27 hemos hecho una aplicación nativa en el fichero `ejecuta.c` que ejecuta una máquina virtual Java y después ejecuta la función `main()` de una clase que le pasamos como argumento, que es lo que viene a hacer el comando `java`.

```
# include <jni.h>

int main(int argc, char* argv[])
{
 JNIEnv* env;
 JavaVM* jvm;
 jint retorno;
 jclass clase_ejecutar;
 jmethodID main_methodID;
 #ifdef JNI_VERSION_1_2
   JavaVMInitArgs vm_args;
   vm_args.version=JNI_VERSION_1_2;
   vm_args.nOptions = 0;
   vm_args.options = NULL;
   vm_args.ignoreUnrecognized = JNI_FALSE;
   retorno = JNI_CreateJavaVM(&jvm,(void**)&env,&vm_args);
 #else
   JDK1_1InitArgs vm_args;
   vm_args.version = JNI_VERSION_1_1;
   retorno = JNI_GetDefaultJavaVMInitArgs(&vm_args);
   if (retorno<0)
      {
       printf("La libreria no soporta crear"
              " una maquina virtual de la 1.1");
       return 1;
      }
   retorno = JNI_CreateJavaVM(&jvm,&env,&vm_args);
 #endif
 if (retorno<0)
    {
     fprintf(stderr
       ,"No se pudo crear la máquina virtual\n");
     return 1;
    }
 if (argc!=2)
    {
     fprintf(stderr
        ,"Indique clase a ejecutar como argumento\n");
     return 1;
    }
```

```
 clase_ejecutar = (*env)->FindClass(env,argv[1]);
 if (clase_ejecutar==NULL)
    goto acabar;
 main_methodID = (*env)->GetStaticMethodID(env
     ,clase_ejecutar,"main","([Ljava/lang/String;)V");
 if (main_methodID==0)
    goto acabar;
 (*env)->CallStaticVoidMethod(env,clase_ejecutar
                             ,main_methodID,NULL);

 acabar:
 if ((*env)->ExceptionOccurred(env))
    (*env)->ExceptionDescribe(env);
 (*jvm)->DestroyJavaVM(jvm);
}
```

Listado 27: Programa que crea una máquina virtual

Para enlazar las llamadas a la máquina virtual debemos hacerlo con una librería determinada tal como se explicó en el apartado 2.2.1 .

Nota: Si lo vamos a ejecutar en Windows hay que tener cuidado de no copiar el fichero `jvm.dll` al directorio donde creamos el ejecutable, porque `jvm.dll` llama a su vez a otras `dll` que si no se encuentran producen que falle la llamada a `JNI_CreateJavaVM()`, sino que debemos modificar el `PATH` para que apunte al directorio donde está `jvm.dll` instalada.

2.4. Obtener la máquina virtual del proceso

Podemos obtener una referencia a todas las máquinas virtuales de un proceso usando la función:

```
jint JNI_GetCreatedJavaVMs(JavaVM** vmBuf, jsize bufLen, jsize*
nVMs);
```

A la función la pasamos en el parámetro `vmBuf` un array de punteros a JavaVM donde nos debe depositar las máquinas virtuales del proceso. El parámetro `bufLen` indica el tamaño máximo de este buffer. El tamaño debe ser suficiente para que no se llene. La función retorna en el parámetro `nVMs` el número de máquinas virtuales que nos ha devuelto. Actualmente Java sólo soporta una maquina virtual por proceso. Luego la función siempre nos devolverá `1` ó `0` máquinas virtuales.

2.5. La interface `JavaVM`

`JavaVM` es un puntero a una estructura cuyo primer campo es un puntero a una tabla de puntero a funciones. Su estructura se muestra en la Figura 7 es decir, sigue el mismo diseño que el interfaz `JNIEnv` de la Figura 2.

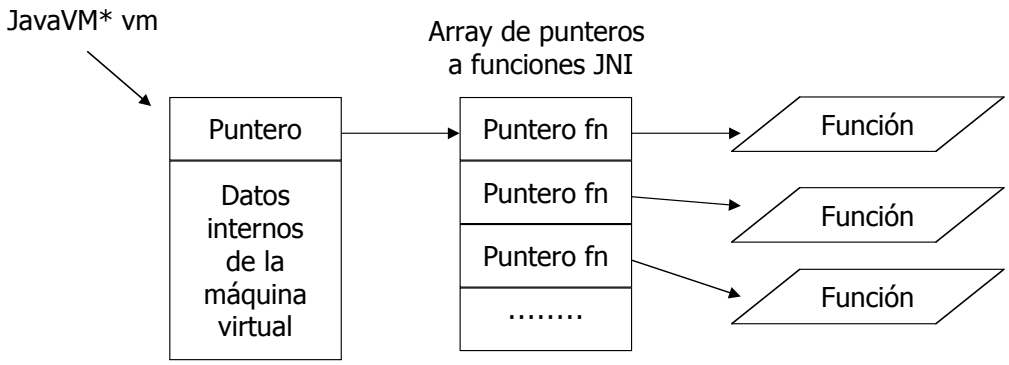

Figura 7: La Interfaz JavaVM

También, al igual que `JNIEnv` las tres primeras entradas de la tabla de punteros a funciones valen `NULL` para una futura compatibilidad con COM. A continuación hay cuatro entradas para las cuatro funciones que podemos ejecutar desde aquí:

```
DestroyJavaVM()                    AttachCurrentThread()
GetEnv()                           DetachCurrentThread()
```

A diferencia del puntero `JNIEnv`, que hay uno por cada hilo, el puntero `JavaVM` existe uno sólo para toda la máquina virtual, y es válido para todos los hilos de la máquina virtual. La función:

```
jint DestroyJavaVM(JavaVM* vm);
```

Descarga la máquina virtual y reclama todos los recursos. El hilo que llama a esta función queda bloqueado hasta que es el único que queda vivo. Esta restricción existe porque los demás hilos pueden tener asignados recursos (p.e. una ventana). En ek JSDK 1.1 sólo el hilo main podía llamar a esta función, la función se bloqueaba hasta que acababan todos los hilos y se liberaban los recursos. Por último la función retornaba devolviendo el código de error `JNI_ERR`. En el JSDK 1.2 se cambió para que cualquier hilo pudiera llamar a esta función. El hilo que la llamaba quedaba bloqueado hasta que los demás hilos acabasen, liberaba los recursos y terminaba. Aunque la función acababa bien, terminaba retornando el código de error `JNI_ERR`. En el JSDK 1.3 la función hace lo mismo que en el JSDK 1.2, pero retorna que ha terminado bien (retorna `JNI_OK`). La función:

```
jint GetEnv(JavaVM* vm, void** penv, jint interface_id);
```

Sirve para obtener el puntero `JNIEnv` del hilo actual a partir del puntero `JavaVM`. El parámetro `interface_id` puede tomar los valores `JNI_VERSION_1_1` o `JNI_VERSION_1_2`, indicando la versión. Las funciones `AttachCurrentThread()` y `DetachCurrentThread()` las veremos en el siguiente punto.

2.6. Asociar hilos nativos a la máquina virtual

En principio sólo el hilo nativo que creó la máquina virtual tiene acceso a ella. La máquina virtual puede tener más hilos, pero deben ser instanciados con

`java.lang.Thread`. Esto es así porque cada hilo que accede a la máquina virtual tiene asociadas dos informaciones:

- Un puntero `JNIEnv` con información asociada a ese hilo.
- Un objeto `java.lang.Thread` que sirve de panel de control para ese hilo.

Si estamos haciendo una aplicación nativa multihilo, cuyos hilos quieren luego acceder a la máquina virtual tenemos un problema, ya que carecen del puntero `JNIEnv` y del objeto `java.lang.Thread` correspondiente. P.e. imaginemos que estamos implementando un servidor web en C cuyos hilos deben ejecutar servlets de una máquina virtual Java. Si desde un hilo nativo intentamos llamar a cualquier función de JNI se producirá un error. Para solucionarlo podemos usar la función:

```
jint AttachCurrentThread(JavaVM* vm, void** penv, void* args);
```

Que asocia al hilo nativo que la ejecuta con una instancia de la máquina virtual. Una vez asociado, el hilo nativo tendrá un objeto de tipo `java.lang.Thread`, y un puntero `JNIEnv` que obtiene en `penv`, con lo que ya podrá ejecutar cualquier función JNI. En el JSDK 1.1 el parámetro `args` siempre debía de ser `NULL`, y al hilo se le asignaba un nombre aleatorio, p.e. "`Thread-123`". Además el hilo siempre pertenecía al `ThreadGroup` "main". En el JSDK 1.2 el parámetro `args` podía valer `NULL` por compatibilidad con JSDK 1.1, pero también podía ser un puntero a una estructura como esta:

```
typedef struct{
    jint version;
    jint char* name;
    jobject group;
} JavaVMAttachArgs;
```

Donde el campo `version` debe valer `JNI_VERSION_1_2`. El campo `name` puede ser `NULL` o apuntar a un string UTF-8 con el nombre del hilo. Si `name` vale `NULL` se le asigna un nombre aleatorio. El campo `group` puede valer `NULL`, en cuyo caso se asigna el hilo al `ThreadGroup` "main", o indicar en una referencia global el `ThreadGroup` al que queremos que pertenezca. Al acabar podemos desasociar el hilo nativo de la máquina virtual con:

```
jint DetachCurrentThread(JavaVM* jvm);
```

También podemos usar la función:

```
jint GetEnv(JavaVM* vm, void** penv, jint interface_id);
```

Para preguntar si un hilo está asociado a la máquina virtual, ya que devuelve un puntero `JNIEnv` que vale `NULL` si el hilo no está asociado a la máquina virtual.

2.7. Ejecutar rutinas al cargar y descargar la máquina virtual

En JSDK 1.2 se ha añadido la posibilidad de indicar rutinas que queramos ejecutar cuando la máquina virtual cargue o descargue nuestra librería de enlace dinámico.

Para ello, en la librería de enlace dinámico donde tengamos implementados nuestros métodos nativos tenemos que exportar dos funciones:

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* jvm, void* reserved);
JNIEXPORT void JNICALL JNI_OnUnload(JavaVM* jvm, void* reserved);
```

`JNI_OnLoad()` se ejecuta cuando `System.loadLibrary()` carga la librería nativa, en ese momento mira si existe está función exportada en la librería, en cuyo caso la ejecuta. En la implementación de esta función ya podemos llamar a cualquiera de las funciones de JNI. Un uso típico de esta función es cachear el puntero `JavaVM`. En la implementación de esta función debemos retornar `JNI_ERR` si nuestra función encuentra una razón por la que la carga de la librería no debe continuar, y la máquina virtual aborta la carga de la librería produciendo una excepción. En caso contrarió debemos retornar la versión, que normalmente será `JNI_VERSION_1_2`. La función:

```
JNIEXPORT void JNICALL JNI_OnUnload(JavaVM* jvm, void* reserved);
```

Es ejecutada por la máquina virtual cuando descarga la librería. Pero vamos a detallar cuales son las reglas para descargar una librería:

¿Cuándo descarga la librería la máquina virtual?

La máquina virtual asocia cada librería nativa que carga con el `ClassLoader` de la clase que llama a `System.loadLibrary()`. Una vez cargada la máquina virtual no la descarga hasta que ve que el `ClassLoader` que la cargo ya no es útil, esto implica que ninguna de las clases que cargo el `ClassLoader` son útiles ya, incluida, como no, la clase que cargo la librería.

¿Qué hilo ejecuta este método?

A `JNI_OnLoad()` lo llama el hilo que llamó a `System.loadLibrary()`, pero a `JNI_OnUnload()` se le llama o bien síncronamente al llamar a `System.runFinalization()`, o bien asíncronamente al final de la vida de la máquina virtual. Con lo que no sabemos que hilo lo ejecutará, en consecuencia no se recomienda poner operaciones de sincronización complejas que puedan llevar a un deadlock. Una última consideración es que `JNI_OnUnload()` se ejecuta cuando las clases del `ClassLoader` ya no son válidas, con lo que no debemos hacer referencia a estas clases en la implementación de esta función.

3. Carga y enlace de librerías de enlace dinámico desde Java

Antes de que la aplicación Java pueda ejecutar un método nativo, la máquina virtual debe buscar y cargar la librería de enlace dinámico donde está implementado el método. En este apartado vamos a comentar más detenidamente cómo se realiza este proceso.

3.1. Los class loader

Los **class loader** están representados en Java por la clase `ClassLoader`. La principal función de estos objetos es cargar las clases Java en la memoria de la máquina virtual, así como comprobar la integridad de las clases. Dado el nombre de una clase, la misión del `ClassLoader` es cargar una clase en la memoria de la máquina virtual. Normalmente la máquina virtual tiene un único `ClassLoader`, llamado **class loader de sistema**, cuya misión es cargar las clases del `CLASSPATH`.

La máquina virtual puede definir objetos class loaders adicionales para cargar otras clases de otro sitio distinto, p.e. un `ClassLoader` para cargar las clases de un applet a través de una conexión HTTP o las clases de un sistema distribuido como RMI o CORBA. Normalmente las clases cargadas por estos `ClassLoader` tienen permisos más restringidos que los del `ClassLoader` de sistema.

Los `ClassLoader` siguen un modelo jerárquico, donde cada `ClassLoader` antes de cargar una clase pide la clase a su `ClassLoader` padre, y sólo si éste no la puede cargar, la carga él. Como muestra la Figura 8, en la raíz de esta jerarquía esta el `ClassLoader` de sistema.

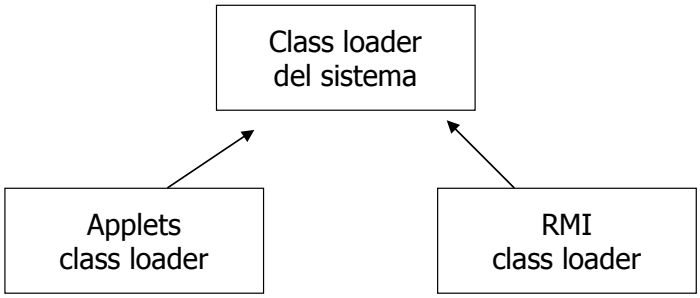

Figura 8: Jerarquía de class loaders

Una segunda utilidad de los class loaders es localizar y cargar las librerías de enlace dinámico. En este tutorial suponemos que el lector está familiarizado con los `ClassLoader` y cómo se usan para cargar bytecodes Java, luego nos vamos a centrar en explicar sólo la segunda utilidad.

3.2. Carga de librerías nativas por el class loader

Al arrancar la máquina virtual, ésta crea una lista de directorios que se usarán para buscar librerías de enlace dinámico nativas. Como explicamos en el apartado 3 de la Parte I, cada SO sigue sus reglas para buscar estás librerías, y la máquina virtual Java sigue el mismo criterio que el SO donde se está ejecutando, tal como indica la Tabla 8.

Sistema Operativo	Regla de búsqueda de librerías de enlace dinámico
Mac OS X	Busca en el directorio donde esta la aplicación y en la variable de entorno `DYLD_LIBRARY_PATH`
UNIX	Busca en los directorios indicados en la variable de entorno `LD_LIBRARY_PATH`
Windows	Busca en el directorio donde esta la aplicación (`.exe`) y en los directorios indicados por la variable de entorno `PATH`

Tabla 8: Reglas para buscar librerías de la máquina virtual Java

Esta ruta de directorios se puede obtener preguntando por la propiedad de sistema `java.library.path`. También podemos modificar esta propiedad para modificar la lista de directorios donde la máquina virtual buscará las librerías. Por ejemplo podemos modificar esta propiedad al lanzar la máquina virtual pasando esta propiedad del sistema al comando `java` así:

```
$ java –Djava.library.path=/usr/lib:. MiPrograma
```

Como el nombre y la ruta del fichero de librería varía en cada plataforma, tenemos los métodos:

```
static String <System> mapLibraryName()
```

Que a partir del nombre independiente de la plataforma obtiene su correspondiente nombre dependiente de la plataforma. P.e. si le pasamos el nombre "`HolaMundo`" nos devolvería "`libHolaMundo.jnilib`" en Mac OS X y nos devolverá "`HolaMundo.dll`" en Windows. Una vez que sabemos el nombre dependiente de la plataforma usaríamos:

```
String <ClassLoader> findLibrary(String name)
```

Para que a partir del nombre del fichero (p.e. `libHolaMundo.jnilib`) Java nos devuelva su ruta absoluta o `null` si no encuentra el fichero en la lista de directorios donde busca. P.e. a partir del nombre "`libHolamundo.jnilib`" nos podría devolver el nombre "`/Users/fernando/JNI/libHolaMundo.jnilib`". Una vez que el `ClassLoader` ya tiene esta ruta, ya puede cargar la librería de enlace dinámico realizando llamadas nativas del SO, y cuando el class loader ya tenga cargada la librería, ésta queda cargada hasta que el recogedor de basura libere al class cloarder que cargó esa librería, momento en el cual también se descargan las librerías de enlace dinámico cargadas por ese class loader.

3.3. Enlace de los métodos nativos

La máquina virtual (y no el `ClassLoader`) es la encargada de enlazar cada método nativo Java con su correspondiente función en las librerías de enlace dinámico. De esta forma cuando llamamos a un método nativo Java, la máquina virtual determina qué función de librería debe de ejecutar. P.e. si tenemos la clase:

```
public class HolaMundo
{
 private native void saluda();
 ..............
}
```

La máquina virtual debe saber que tiene que llamar a la función de librería `Java_HolaMundo_saluda()` cuando ejecutamos el método `saluda()`. La pregunta es: ¿Cómo sabe la máquina virtual cómo se llama, y en qué fichero está la función de librería que tiene que ejecutar cuando llamemos al método nativo Java?. Para saber cómo se llama la función de librería a ejecutar correspondiente al método nativo, Java sigue una regla a la hora de nombrar a los métodos nativos, de esta forma la máquina virtual puede deducir el nombre de la función a partir del nombre del método. Para obtener el nombre de la función correspondiente a un método nativo se sigue las siguientes reglas:

1. Precedemos el nombre por `Java_`
2. Se añade el nombre de la clase
3. Se añade otro _
4. Se añade el nombre del método
5. Sólo en el caso de los métodos sobrecargados se añaden dos __ precedidos por el nombre del tipo de los parámetros

Por ejemplo para el método nativo anterior el nombre de la función de librería debería llamarse: `Java_HolaMundo_saluda()`. Si en vez de definirla así, tenemos el método sobrecargado:

```
public class HolaMundo
{
 private native void saluda();
 private native void saluda(String mensaje);
 ............
}
```

Entonces se usaría también el nombre de los tipos de los parámetros y las funciones se deberían llamar respectivamente:

```
Java_HolaMundo_saluda__()
Java_HolaMundo_saluda__Ljava_lang_String_2()
```

Como los nombres de los métodos Java pueden contener cualquier carácter Unicode, y las funciones de librería suelen sólo poder tener un nombre ASCII, los nombres de los métodos se codifican según un encoded name con caracteres de escape aprovechando el hecho de que los nombres de los métodos deben empezar por una letra, con lo que se utilizan las secuencias de escape `_0` hasta `_9` según muestra la

Tabla 9. Siguiendo estas reglas la máquina virtual ya puede saber el nombre de la función a ejecutar para un determinado método nativo.

Secuencia de escape	Representa
`_0xxxx`	Un carácter Unicode en hexadecimal puesto en `xxxx`
`_1`	El carácter _
`_2`	El carácter `;`
`_3`	El carácter `[`

Tabla 9: Secuencias de escape para nombres de símbolos Java

Queda por ver cómo la máquina virtual busca la libraría de enlace dinámico donde está implementado el método. Para ello la máquina virtual sigue los siguientes pasos:

1. Determina cuál es el `ClassLoader` de la clase que tiene el método nativo.

Para ello cada clase tiene el método:

```
ClassLoader <Class> getClassLoader()
```

que nos devuelve el `ClassLoader` que cargó una clase.

2. Busca por la lista de librerías de enlace dinámico cargadas por ese `ClassLoader`, para ver cuál es la función que implementa ese método nativo.

3. Fija unas estructuras internas para que la próxima vez que se llame a este método nativo se llame directamente a la función de librería sin tener que buscar el `ClassLoader`.

La máquina virtual no realiza el enlace del método nativo con su correspondiente función de librería hasta que no se llama por primera vez al método. De esta forma se consiguen dos beneficios: Por un lado no se enlazan llamadas a métodos que no se usan, y por otro lado se evitan posibles errores de enlace debidos a que la librería donde está implementada la función de librería del método nativo no ha sido cargada todavía, es decir, no se ha llamado todavía a `System.loadLibrary()` para la librería donde está implementado la función del método nativo.

3.4. Enlace manual de métodos nativos

Normalmente el enlace entre un método nativo y su correspondiente función de librería lo realiza la máquina virtual de forma automática, tal como acabamos de explicar. Si por alguna razón no quisiéramos emparejar un método nativo con una función que no siguiese estas reglas para su nombre podemos usar la función de JNI:

```
jint RegisterNatives(JNIenv* env, jclass class, const
JNINativeMethod* methods, jint nMethods);
```

que nos permite indicar esta correspondencia. El parámetro `class` es la clase a la que registrar los métodos nativos, el parámetro `methods` es un array de elementos

del tipo `JNINativeMethod` con información de los métodos a registrar, y el parámetro `nMethods` es el número de métodos que hay en el array. Además `JNINativeMethod` es una estructura con los siguientes campos:

```
typedef struct{
  char* name;
  char* signature;
  void* fnPtr;
} JNINativeMethod;
```

Donde el campo `name` es el nombre de la función. El campo `signature` indica el tipo de los parámetros de la función, y el campo `fnPtr` es un puntero a la función a registrar. P.e. para registrar al método nativo:

```
private native void <HolaMundo> saluda(String mensaje);
```

del ejemplo anterior haríamos:

```
JNINativemethod nm;
nm.name="saluda";
nm.signature="(Ljava/lang/String;)V";
nm.fnPtr = mi_funcion_c
(*env)->RegisterNatives(env,clase,&nm,1);
```

Obsérvese que de esta forma no es necesario exportar a la función `mi_funcion_c()` con el modificador `JNIEXPORT`, pero lo que sí que sigue siendo necesario es que siga el modelo de llamada de JNI, es decir que la función tenga el modificador `JNICALL`. ¿En qué casos es recomendable realizar manualmente este enlace?. Pues básicamente en tres casos:

- Cuando una aplicación nativa tiene una máquina virtual incrustada, la máquina virtual puede no poder encontrar las funciones a las que llamar porque ésta sólo busca en las librerías nativas, y no en la aplicación misma.
- Podemos usar `RegisterNatives()` varias veces para actualizar la función a la que llama un método nativo, con el fin de actualizar la función a la que llamar en tiempo de ejecución.
- En situaciones de tiempo real es más conveniente dejar enlazados un largo número de métodos, que esperar a que la máquina virtual los enlace a posteriori.

4. Compatibilidad y acceso a librerías del host environment desde Java

Con el fin de facilitar la migración a Java desde otros lenguajes sería deseable que una aplicación Java pudiera acceder a librerías nativas escritas en otros lenguaje de programación tradicional como C o C++. De esta forma el programador no tiene que volver a hacerse librerías de programación que ya tenía hechas. En este apartado vamos a comentar cómo se consigue acceder a estas librerías desde Java. El principal problema con el que se va a encontrar el programador es que estas funciones no tienen porque tener los parámetros que siempre reciben las funciones de librería para JNI (puntero `JNIEnv*` y referencia `jobject`), ni tampoco tienen porque seguir el mismo mecanismo de paso de parámetros.

Existen tres técnicas para conseguir acceder estas a funciones de librería escritas para otros lenguajes, que vamos a ir explicando a lo largo de este apartado: Función de stub, shared stubs y objetos peer.

4.1. Mecanismos de paso de parámetros

El mecanismo de paso de parámetros determina en qué orden se ponen los parámetros de la función llamada en la pila, así como que mecanismos se siguen para retornar un valor. JNI no sigue el mismo mecanismo de paso de parámetros en todas las plataformas: En los sistemas UNIX incluido Mac OS X se siguen el mecanismo de paso de parámetros de C y C++, también llamado `__cdecl`. En Windows se sigue el mecanismo de paso de parámetros que sigue la API de Windows `__stdcall`. También conocido como mecanismo de paso de parámetros de Pascal. En cualquier caso nosotros debemos declarar a las funciones siempre con el modificador `JNICALL`, el cual se sustituye por el mecanismo que se utilice realmente en le sistema donde estemos. Es decir en UNIX este modificador está declarado como:

```
#define JNICALL
```

Mientras que en Windows está declarado como:

```
#define JNICALL __stdcall
```

Cuando queramos llamar a una función que no siga el mismo mecanismo de paso de parámetros necesitaremos escribir lo que llamaremos una **función de stub** que transforme el paso de parámetros de un mecanismo a otro, como vamos a ver en el siguiente apartado.

4.2. Función de stub

Cuando queramos llamar a una función de librería que no cumpla con las reglas que siguen las funciones de librería de los métodos nativos, una primera solución puede ser implementar nosotros una función de librería asociada a un método nativo Java que sí se adapte a estas reglas, y llamar desde esta función a la función de librería a

la que queremos acceder. P.e supongamos que queremos acceder desde Java a la función:

```
int atoi(const char* str);
```

En este caso podríamos implementar un método nativo así:

```
public class Utilidad
{
 public static native int atoi(String str);
 ...........
}
```

Y después desde una función de librería recoger los parámetros de la llamada y pasárselos a `atoi()` así:

```
JNIEXPORT jint JNICALL Java_Utilidad_atoi(JNIEnv* env, jclass clase,
jstring str)
{
 int resultado;
 const char* c_str = (*env)->GetStringUTFChars(
                                        env,str,NULL);
 if (c_str==NULL)
    return 0; // OutOfMemoryException
 resultado = atoi(c_str);
 (*env)->ReleaseStringUTFChars(env,str,c_str);
 return resultado;
}
```

4.3. Shared Stubs

El mecanismo que acabamos de ver en el apartado anterior tiene el inconveniente de que por cada función nativa a la que queramos acceder tenemos que implementar su correspondiente función de stub, lo cual se vuelve tedioso cuando queremos acceder a gran número de funciones nativas. En este apartado vamos a ver un mecanismo llamado **shared stubs**, que nos permite automatizar el proceso de creación de funciones de stub con el fin de simplificar el proceso de creación de clases wrapper sobre librerías nativas. Un shared stub es un objeto Java que se va a encargar de llamar a una función nativa, cuyo nombre, parámetros y mecanismo de llamada le diremos previamente al objeto. Antes de describir detalladamente la clase Java `FuncionC` que actuará como shared stub, vamos a ver cómo se haría ahora para llamar a `atoi()` desde Java:

```
try{FuncionC atoi = new FuncionC("msvcrt.dll", "atoi",
                                  FuncionC.PARAMETROS_C);
    int i = atoi.ejecutaFuncionInt(new Object[] {"45"});
    System.out.println("El numero convertido es:"+i);
   }
catch (UnsatisfiedLinkError ex){
    ex.printStackTrace();
   }
```

En el constructor del objeto `FuncionC` le decimos qué fichero de librería, nombre de función y mecanismo de paso de parámetros tiene la función a ejecutar. Después le

pedimos al objeto que ejecute la función, e indicamos en un array de elementos de tipo `Object` los parámetros de la función.

Podemos diseñar una jerarquía de clases como la que se muestra en la Figura 9 donde `PunteroC` represente un puntero C genérico y que tenga dos derivadas: `MallocC` que representa un puntero C apuntando a un trozo de memoria asignada con `malloc()`, y `FuncionC` que represente un puntero a función C. Esta jerarquía de clases la vamos a implementar en el apartado 4.5.

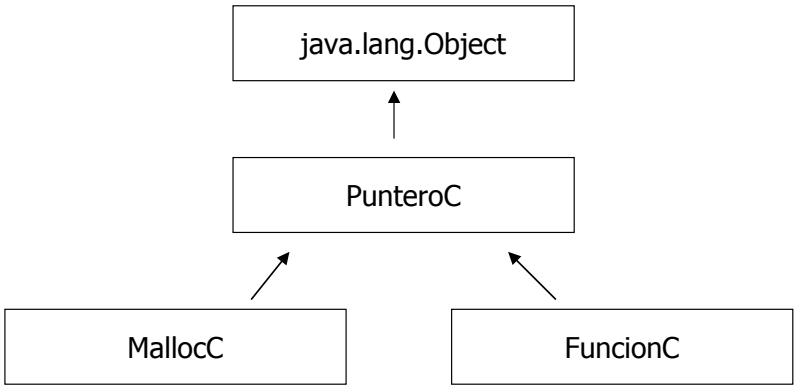

Figura 9: Jerarquía de clases para los shared stubs

4.4. Diferencias entre funciones de stub y shared stubs

Entre las dos formas que hemos visto de acceso a funciones nativas, cada una tiene sus ventajas y sus inconvenientes: La principal ventaja de los shared stubs es que una vez hecha la clase `FuncionC`, el programador no necesita escribir una sola línea de código C para acceder a funciones nativas. Pero los shared stubs se deben usar con cuidado, ya que el programador Java está usando funciones C desde Java, con lo que un error ejecutando una función C puede corromper la memoria o hacer fallar a toda el programa Java. La principal ventaja de las funciones de stub es que son más eficientes en el paso de parámetros que los shared stubs. En la práctica el programador tendrá que elegir entre eficiencia o reducción de los plazos de entrega. Las funciones de stub se recomiendan usar cuando vamos a hacer librerías de uso general que queramos optimizar al máximo, en caso contrarío es más cómodo usar un shared stub.

4.5. Implementación de los shared stubs

Vamos a crear una implementación de las clases `PunteroC`, `MallocC` y `FuncionC` anteriores. Estas clases nos permitirán trabajar con elementos de C desde Java. Hay que tener en cuenta que esta librería de clases es peligrosa, ya que el programador Java tiene la posibilidad de usar cualquier elemento de C accediendo a direcciones de memoria o ejecutando funciones de forma arbitraría. El uso de estas librerías se debería limitar usando un `SecurityManager` para que código inseguro no pueda acceder a ellas.

4.5.1 La clase `PunteroC`

Esta clase la vamos a diseñar como muestra el Listado 28. Vemos que la clase `PunteroC` tiene un atributo llamado `puntero` de tipo entero y que contiene la dirección de memoria C a la que hace referencia nuestro objeto.

```
public abstract class PunteroC
{
 protected int puntero=0;
 public int getPuntero()
       {
        return puntero;
       }
 public void setPuntero(int pos)
       {
        puntero = pos;
       }
 public native byte getByte(int offset);
 public native void setByte(int offset,int valor);
}
```

Listado 28: Implementación de `PunteroC`

Después la vamos a poner métodos de acceso nativos `getByte()`, `setByte()` que nos permitan leer o escribir los datos de la dirección de memoria a la que apunta el puntero. Estos métodos estarán implementados en C tal como muestra el Listado 29.

```
JNIEXPORT jbyte JNICALL Java_PunteroC_getByte(JNIEnv* env,
jobject obj, jint offset)
{
 char* puntero;
 jclass clase = (*env)->GetObjectClass(env,obj);
 jclass clase_excepcion =
  (*env)->FindClass(env,"java/lang/UnsatisfiedLinkError");
 jfieldID punteroID =
       (*env)->GetFieldID(env,clase,"puntero","I");
 if (punteroID==0)
  {
   (*env)->ThrowNew(env,clase_excepcion
             ,"No se encuentra el atributo puntero");
    return 0;
  }
 puntero = (byte*)((*env)->GetIntField(env,obj,punteroID));
 return (puntero[offset]);
}

JNIEXPORT void JNICALL Java_PunteroC_setByte(JNIEnv* env,
jobject obj, jint offset, jint valor)
{
 char* puntero;
 jclass clase = (*env)->GetObjectClass(env,obj);
 jclass clase_excepcion =
  (*env)->FindClass(env,"java/lang/UnsatisfiedLinkError");
 jfieldID punteroID =
           (*env)->GetFieldID(env,clase,"puntero","I");
```

```
 if (punteroID==0)
  {
   (*env)->ThrowNew(env,clase_excepcion
                ,"No se encuentra el atributo puntero");
    return;
  }
 (*env)->SetIntField(env,obj,punteroID,valor);
}
```

Listado 29: Funciones nativas para métodos getter y setter

4.5.2 La clase `MallocC`

Los objetos de esta clase representan un trozo de memoria que fue asignada con `malloc()` y que puede ser liberada con `free()`. Al crear el objeto indicamos la cantidad de memoria que queremos reservar. El Listado 30 muestra la implementación de esta clase. La implementación de los métodos nativos se muestra en el Listado 31.

```
public class MallocC extends PunteroC
{
 public MallocC(int tamano)
       {
        puntero = malloc(tamano);
       }
 private native int malloc(int tamano);
 public native void free();
}
```

Listado 30: Implementación de la clase MallocC

```
JNIEXPORT jint JNICALL Java_MallocC_malloc(JNIEnv* env,
jobject obj, jint tamano)
{
 return (int)malloc(tamano);
}

JNIEXPORT void JNICALL Java_MallocC_free(JNIEnv* env, jobject
obj)
{
 void* puntero;
 jclass clase = (*env)->GetObjectClass(env,obj);
 jclass clase_excepcion = (*env)->FindClass(env,
                      "java/lang/UnsatisfiedLinkError");
 jfieldID punteroID = (*env)->GetFieldID(env,clase
                                      ,"puntero","I");
 if (punteroID==0)
  {
   (*env)->ThrowNew(env,clase_excepcion
                  ,"No se encuentra el atributo puntero");
    return;
  }
 puntero = (void*)((*env)->GetIntField(env,obj,punteroID));
```

```
 free(puntero);
}
```

Listado 31: Implementación de los métodos nativos de `MallocC`

4.5.3 La clase `FuncionC`

Cada objeto de tipo `FuncionC` va a representar un puntero a función de librería. Al crear el objeto indicaremos:

- El fichero de librería donde está la función implementada
- El nombre de la función
- El mecanismo de llamada a la función: Si es una función de tipo `__cdecl` o de tipo `__stdcall`

El Listado 32 muestra la forma de la clase `FuncionC`. El constructor llama a la función nativa `cargaFuncion()` que carga la función usando las APIs nativas del sistema donde estemos trabajando. Después podremos ejecutar la función con el método `ejecutaFuncionInt()`, el cual recibe como parámetro un array de `Object` con los parámetros y devuelve un entero, que es el resultado de ejecutar la función. Análogamente podríamos implementar métodos que retornasen otros tipos de datos.

```
public class FuncionC extends PunteroC
{
 public static int PARAMETROS_C=0;
 public static int PARAMETROS_STDCALL=1;
 private int pasoParametros;
 public FuncionC(String libreria // Libreria a usar
                ,String nombreFuncion // Nombre funcion
                ,int pasoParametros // PARAMETROS_C o
                                    // PARAMETROS_STDCALL
          ) throws  UnsatisfiedLinkError
        {
         puntero = cargaFuncion(libreria,nombreFuncion);
         this.pasoParametros = pasoParametros;
        }
 private native int cargaFuncion(String libreria
      ,String nombreFuncion) throws UnsatisfiedLinkError;
 public native int ejecutaFuncionInt(Object[] parametros)
                              throws UnsatisfiedLinkError;
}
```

Listado 32: Implementación de la clase `FuncionC`

Como ejemplo, el Listado 33 muestra cómo se hace en Windows para implementar estas funciones nativas. Ya explicamos al hablar de cómo enlazar las llamadas con una máquina virtual desconocida cómo funcionan las APIs de Windows `LoadLibrary()` y `GetProcAddress()` para obtener puntero a funciones de la librearía de enlace dinámico.

```
// Representa una palabra del hardware
typedef union {
    jint i;
    jfloat f;
    void *p;
} word_t;

// Funciones de apoyo
static const char* GetCadenaNativa(JNIEnv *env
                                        , jstring jstr);
static int ejecuta_asm(void* fn, int n_parametros
                , word_t* parametros, int tipo_llamada);

JNIEXPORT jint JNICALL Java_FuncionC_cargaFuncion(
     JNIEnv* env, jobject obj, jstring libreria
   , jstring nombreFuncion)
{
 void* fn;
 HANDLE hLibreria;
 const char* str_libreria = GetCadenaNativa(env,libreria);
 const char* str_nombre_funcion =
    GetCadenaNativa(env,nombreFuncion);
 jclass clase_excepcion =
  (*env)->FindClass(env,"java/lang/UnsatisfiedLinkError");
 // Intenta cargar la librerÌa
 str_libreria = GetCadenaNativa(env,libreria);
 hLibreria = LoadLibrary(str_libreria);
 if (hLibreria==0)
  {
   (*env)->ThrowNew(env,clase_excepcion
                    ,"No se encuentra la libreria");
   goto fin;
  }

 // Busca la funcion en la libreria
 fn = GetProcAddress(hLibreria,str_nombre_funcion);
 if (fn==NULL)
   (*env)->ThrowNew(env,clase_excepcion
                        ,"No se encuentra la funcion");

 // Libera memoria
 fin:
 free(libreria);
 free(nombreFuncion);
 return (int)fn;
}

#define MAX_PARAMS 32

JNIEXPORT jint JNICALL Java_FuncionC_ejecutaFuncionInt(JNIEnv*
env
           , jobject bj, jobjectArray array_parametros)
{
 jnt retorno;
```

```
int n_parametros;
int i;
int tipo_llamada;
jboolean es_cadena[MAX_PARAMS];
word_t parametros[MAX_PARAMS];
jclass tipo_entero =
      (*env)->FindClass(env,"java/lang/Integer");
jclass tipo_boolean =
      (*env)->FindClass(env,"java/lang/Boolean");
jclass tipo_float =
      (*env)->FindClass(env,"java/lang/Float");
jclass tipo_puntero =
      (*env)->FindClass(env,"PunteroC");
jclass tipo_string =
      (*env)->FindClass(env,"java/lang/String");
jfieldID punterofieldID =
      (*env)->GetFieldID(env,tipo_puntero,"puntero","I");
jfieldID tipollamadafieldID;
void* fn;
jclass clase = (*env)->GetObjectClass(env,obj);

n_parametros = (*env)->GetArrayLength(env
                                  ,array_parametros);
if (n_parametros>MAX_PARAMS)
 {
    jclass clase_excepcion = (*env)->FindClass(
               env,"java/lang/UnsatisfiedLinkError");
  (*env)->ThrowNew(env,clase_excepcion
             ,"No se encuentra la funcion");
  return 0;
 }

// Convierte los par·metros de Java a C
for(i=0;i<n_parametros;i++)
 {
  jobject param = (*env)->GetObjectArrayElement(
                                 env,array_parametros,i);
  es_cadena[i] = JNI_FALSE;
  if (param==NULL)
    parametros[i].p=NULL;
  else if ((*env)->IsInstanceOf(env,param,tipo_entero))
   {
    jmethodID methodID = (*env)->GetMethodID(
                   env,tipo_entero, "hashCode", "()I");
    parametros[i].i=(*env)->CallIntMethod(
                                 env,param,methodID);
   }
  else if ((*env)->IsInstanceOf(
                           env,param,tipo_boolean))
   {
    jmethodID methodID = (*env)->GetMethodID(
              env,tipo_boolean, "booleanValue", "()Z");
    parametros[i].i=(*env)->CallBooleanMethod(
                                  env,param,methodID);
   }
```

```
   else if ((*env)->IsInstanceOf(env,param,tipo_float))
    {
     jmethodID methodID = (*env)->GetMethodID(
                  env,tipo_float, "floatValue", "()F");
     parametros[i].f=(*env)->CallFloatMethod(
                                      env,param,methodID);
    }
   else if ((*env)->IsInstanceOf(
                                env,param,tipo_puntero))
    {
     parametros[i].i=(*env)->GetIntField(
                               env,param,punterofieldID);
    }
   else if ((*env)->IsInstanceOf(env,param,tipo_string))
    {
     const char* cstr = GetCadenaNativa(env,param);
     if (cstr==NULL)
       goto fin; // Se produjo una excepcion
        parametros[i].p = (void*)cstr;
     es_cadena[i] = JNI_TRUE;
    }
     else
    {
     jclass clase_excepcion = (*env)->FindClass(env
                        ,"java/lang/UnsatisfiedLinkError");
     (*env)->ThrowNew(env,clase_excepcion
                               ,"Tipo de dato no valido");
     goto fin;
    }
   (*env)->DeleteLocalRef(env,param);
  }

 // Llamamos a la funcion
 fn = (void*)(*env)->GetIntField(env,obj,punterofieldID);
 tipollamadafieldID = (*env)->GetFieldID(
                           env,clase,"pasoParametros","I");
 if (tipollamadafieldID==0)
   printf("0\n");
 tipo_llamada = (*env)->GetIntField(
                               env,obj,tipollamadafieldID);
 if ((*env)->ExceptionCheck(env))
     (*env)->ExceptionDescribe(env);
 retorno = ejecuta_asm(fn,n_parametros
                       ,parametros,tipo_llamada);

 fin:
   for(i=0;i<n_parametros;i++)
    {
       if (es_cadena[i])
       free(parametros[i].p);
    }
 return retorno;
}
```

```
static int ejecuta_asm(void* fn, int n_parametros, word_t*
parametros, int tipo_llamada)
{
 __asm{
     mov esi,parametros
     mov edx,n_parametros
     // Transforma n_parametros de palabras a bytes
     shl edx,2
     sub edx,4
     jc puestros_parametros

     // Pone los parametros en la pila
     bucle_parametros:
     mov eax, DWORD PTR[esi+edx]
     push eax
     sub edx,4
     jge SHORT bucle_parametros

     // Llama a la funcion C
     puestros_parametros:
     call fn

     // Comprueba el tipo de llamada
              mov edx,tipo_llamada
     or edx,edx
     jnz llamada_stdcall

     // Saca los argumentos
     mov edx,n_parametros
     shl edx,2
     add esp,edx

     llamada_stdcall:
     // Hecho, eax contiene el retorno
  }
}

static const char* GetCadenaNativa(JNIEnv *env, jstring jstr)
{
 char* resultado;
 jbyteArray chars;
 jint longitud;
 static jmethodID MID_String_getBytes;

 // Obtenemos el methodID del String.getBytes()
 MID_String_getBytes=(*env)->GetMethodID(env
         ,(*env)->FindClass(env,"java/lang/String")
                   , "getBytes", "()[B");
 if (MID_String_getBytes==NULL)
    {
    (*env)->ExceptionDescribe(env);
    return NULL;
    }

 // Lo ejecutamos sobre el String para obtener un array nativo
```

```
 chars = (*env)->CallObjectMethod(env,jstr
                                  , MID_String_getBytes);
 if ((*env)->ExceptionOccurred(env))
    {
     (*env)->ExceptionDescribe(env);
     return NULL;
    }

 // Creamos un buffer donde ponemos el array de caracteres
obtenido
 longitud = (*env)->GetArrayLength(env,chars);
 resultado = (char*)malloc(longitud+1);
 (*env)->GetByteArrayRegion(env,chars, 0, longitud,
(jbyte*)resultado);
 resultado[longitud] = '\0';

 // Liberamos la referencia local al array de bytes
 (*env)->DeleteLocalRef(env,chars);
 return resultado;
}
```

Listado 33: Implementación de los métodos nativos de la clase `FuncionC`

4.6. Objetos peer

Ya hemos visto cómo podemos usar funciones de stub y shared stubs para acceder a funciones C. Otro problema que podemos querer resolver es el de referirnos a estructuras de datos C desde Java. Para referirnos a estas estructuras de datos podemos usar los llamados **objetos peer**, que son estructuras de datos C apuntadas por un objeto Java. La Figura 10 muestra esta idea. Vemos que para implementarlos podemos usar `PunteroC` o derivadas.

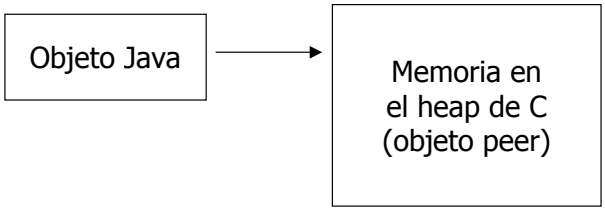

Figura 10: Objeto peer de un objeto Java

Los objetos peer se suelen usar para representar estructuras de datos nativas como descriptores de fichero, descriptores de sockets, ventanas, y otros elementos de interfaz gráfica.

4.6.1 Objetos peer del JSDK

Las implementaciones de la máquina virtual usan objetos peer para implementar paquetes como `java.io.*`, `java.net.*` o `java.awt.*`. P.e. la clase `java.io.FileDescriptor` contiene un atributo privado llamado `fd` que representa el file descriptor del fichero:

```
public final class FileDescriptor
{
 private int fd;
 ..........
}
```

Supongamos que queremos realizar una operación que no está soportada por el API de Java. Podríamos usar JNI para acceder al atributo `fd` — ya que JNI nos permite acceder a los atributos privados de un objeto — y después realizar una operación nativa directamente sobre este descriptor de fichero. Esto sin embargo da lugar a un par de problemas:

1. La operación que ejecutemos directamente sobre el descriptor de fichero puede alterar el estado interno del objeto peer (p.e. si cerramos el fichero), ya que los objetos peer suponen que ellos tienen acceso exclusivo a las estructuras de datos nativas.
2. Nadie nos garantiza que en futuras implementaciones de la máquina virtual ese atributo siga llamándose `fd`, o que siga ahí.

En consecuencia, no debemos usar nunca los atributos de una clase que apuntan a un objeto peer, sino que debemos crearnos nuestras propias clases Java que apunten a sus propios objetos peer.

4.6.2 Liberación de objetos peer

Los objetos Java que apuntan a objetos peer se hacen en lenguaje Java, con lo que los objetos son eliminados por el recogedor de basura automáticamente cuando dejan de necesitarse. Pero ahora necesitamos asegurarnos de que su correspondiente objeto peer sea liberada cuando se libera la clase. P.e. si usamos un objeto de tipo `MallocC` debemos acordarnos de llamar al método `free()` antes de perder la referencia al objeto. Hay programadores que les gusta poner una llamada a la liberación del peer en el método `finalize()`, que es el método que ejecuta la máquina virtual antes de liberar cualquier objeto de la forma:

```
public class MallocC extends PunteroC
{
 ..........
 protected void finalize()
        {
         free();
         super.finalize();
        }
 private native int malloc(int tamano);
 public native synchronized void free();
}
```

En este caso debemos de poner un control en el método `free()` para evitar que se intente liberar la memoria varias veces. Una por parte del programador y otra por parte de `finalize()`. También es conveniente hacer el método `free()` con el modificador `synchronized`, para evitar que se le llame por parte de varios hilos y se produzca algún tipo de race condition.

4.6.3 Punteros a Java desde el objeto peer

Normalmente es el objeto Java el que tiene un puntero al objeto peer, pero a veces puede ser que el objeto peer también tenga un puntero al objeto Java, tal como muestra la Figura 11.

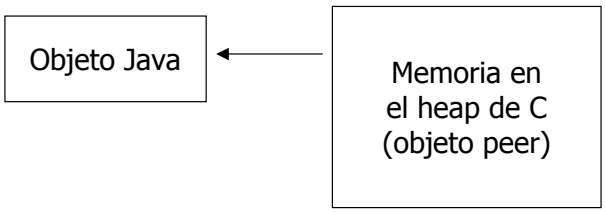

Figura 11: Objeto Java de un objeto peer

En este caso el puntero al objeto Java se suele guardar como una referencia global, en una variable global de C. Esto da lugar al problema de que si liberamos la referencia al objeto Java, el recogedor de basura no puede eliminar el objeto Java, ya que tenemos una referencia C al objeto Java. Para evitarlo se recomienda que el objeto peer mantenga una referencia global desligada al objeto Java que — tal como explicamos en el apartado 7.2.3 — se puede crear con la función:

```
jweak NewWeakGlobalRef(JNIEnv* env, jobject obj);
```

5. Información de introspección desde JNI

La i**nformación de introspección**, también llamada reflection la conseguimos a través de una serie de clases colocadas en el paquete `java.lang.reflection.*` así como algunos métodos colocados en las clases `java.lang.Object` y `java.lang.Class`. Nosotros vamos a suponer que el lector ya conoce estas técnicas y vamos a ver cómo acceder a esta información desde JNI. Para acceder a la información de introspección desde JNI podemos ejecutar los mismos métodos que ejecutaríamos desde Java. Pero además JNI tiene algunas funciones que pueden ayudarnos a realizar operaciones comunes como son:

```
jclass GetSuperclass(JNIEnv* env, jclass class);
```

Que nos devuelve la superclase de una clase o `NULL` si es la clase `java.lang.Object` o una interfaz que no tiene base.

```
jboolean IsAssignableFrom(JNIEnv* env, jclass sourceclass, jclass
targetclass);
```

Nos dice si un objeto de tipo `sourceclass` puede ser apuntado por una referencia del tipo `targetclass`.

```
jclass GetObjectClass(JNIEnv* env, jobject obj);
```

Nos devuelve la clase del objeto `obj`.

```
jboolean IsInstanceOf(JNIenv* env, jobject obj, jclass class);
```

Nos dice si el objeto `obj` es una instancia de la clase `class`.

```
jfieldID FromReflectedField(JNIEnv* env, jobject field);
jobject ToReflectedField(JNIenv* env, jclass class,
                   jfieldID fieldID, jboolean isStatic);
```

Estos métodos nos permiten convertir entre field IDs y objetos de tipo `java.lang.Field`.

```
jmethodID FromReflectedMethod(JNIEnv* env,jobject method);
jobject ToReflectedMethod(JNIEnv* env, jclass class
               ,jmethodID methodID, jboolean isStatic);
```

Estos métodos nos permiten convertir entre method IDs y objetos de tipo `java.lang.Method` o del tipo `java.lang.Constructor`.

6. Programación en C++ con JNI

Este tutorial está escrito para programación en C, pero si vamos a programar en C++ tenemos un interfaz de programación más simple, ya que en vez de hacer llamadas del tipo:

```
jclass clase = (*env)->FindClass(env,"java/lang/String");
```

En C++ hacemos llamadas del tipo:

```
jclass clase = env->FindClass("java/lang/String");
```

Obsérvese que a pesar de la doble indirección del puntero `JNIEnv` (véase Figura 2), en C++ usamos una sola indirección. Esto es porque `JNIEnv` en C++ es una clase cuyos métodos realizan la segunda indirección, es decir si miramos el fichero `jni.h` encontramos:

```
struct JNIEnv_;

#ifdef __cplusplus
   typedef JNIEnv_ JNIEnv;
   struct JNIEnv_ {
       const struct JNINativeInterface_ *functions;
       jint GetVersion() {
           return functions->GetVersion(this);
       }

       jclass FindClass(const char *name) {
           return functions->FindClass(this, name);
       }
    ....
    ....
   }
#endif
```

Es decir, la clase almacena un puntero `functions` que apunta a la tabla de punteros a funciones, y cuando ejecutamos un método, éste usa el puntero para pasar la llamada a la función que corresponda. Además como vemos arriba nos ahorramos pasar como parámetro en la llamada a las funciones el parámetro `env`, ya que éste es el puntero `this` de C++.

Otra ventaja de usar C++ es que el control de tipos es más fuerte que en C. Por ejemplo si usamos C++ el compilador detecta el error típico de pasar una referencia de tipo `jobject` a un parámetro de tipo `jclass`. Esto es así porque los tipos en C++ están creados mediante clases organizadas de forma jerárquica, mientras que en C todos los tipos son redefiniciones del tipo `jobject` hechas con `typedef`, con lo que en C `jobject` y `jclass` son tipos totalmente intercambiables. Es decir si miramos en le fichero `jni.h` encontramos:

```
#ifdef __cplusplus
  class _jobject {};
  class _jclass : public _jobject {};
  class _jthrowable : public _jobject {};
```

```
  class _jstring : public _jobject {};
  class _jarray : public _jobject {};
  .....
  typedef _jobject *jobject;
  typedef _jclass *jclass;
  typedef _jthrowable *jthrowable;
  typedef _jstring *jstring;
  typedef _jarray *jarray;
  .....
#else
  struct _jobject;
  typedef struct _jobject *jobject;
  typedef jobject jclass;
  typedef jobject jthrowable;
  typedef jobject jstring;
  typedef jobject jarray;
#endif
```

Esta comprobación de tipos más estricta que usa C++ a veces necesita añadir castings que no se necesitaban en C. P.e. en C podemos hacer:

```
jstring jstr=(*env)->GetObjectArrayElement(env,array,i);
```

Mientras que en C++ tendríamos que hacer:

```
jstring jstr=(jstring)env->GetObjectArrayElement(array,i);
```

Una última consideración importante es que no existen diferencias en el rendimiento entre C y C++, como muchas veces alegan equivocadamente los programadores de C.

7. Internacionalización con JNI

La máquina virtual Java utiliza Unicode, con lo que puede representar caracteres de cualquier lugar del mundo, pero después cada sistema operativo utiliza su juego de caracteres para representar los caracteres. P.e. Windows, Linux o Mac OS X suelen usar el juego de caracters ISO-8859-1, Mac OS Classic usa el juego de caracteres Mac OS Roman, y BeOS suele utilizar UTF-8. En JNI encontramos dos juegos de funciones para convertir texto Java a texto nativo: `GetStringChars()` que nos devuelve texto Unicode y `GetStringUTFChars()` que nos devuelve texto en UTF-8. Pero como vamos a ver en breve, el uso de estas funciones sólo es recomendable si nuestro sistema operativo trabaja o bien en Unicode o bien en UTF-8, en caso contrario tendremos que convertir a/desde el juego de caracteres nativo de nuestro sistema operativo.

7.1. Sistemas operativos que soportan Unicode

Algunos SO como Mac OS X o Windows NT tienen extensiones que les permiten aceptar texto Unicode en sus APIs. En este caso es posible usar la función `GetStringChars()` directamente. Window NT permite escribir programas tanto usando caracteres ASCII ISO-8859-1 como usando Unicode, para ello Windows NT define en el fichero `tchar.h` los tipos:

```
// Generic types

#ifdef UNICODE
    typedef wchar_t TCHAR;
#else
    typedef unsigned char TCHAR;
#endif

typedef TCHAR * LPTSTR, *LPTCH;

// 8-bit character specific

typedef unsigned char CHAR;
typedef CHAR *LPSTR, *LPCH;

// Unicode specific (wide characters)

typedef unsigned wchar_t WCHAR;
typedef WCHAR *LPWSTR, *LPWCH;
```

Obsérvese que `TCHAR` y `LPTSTR` usaran Unicode o ASCII dependiendo de si está definido el identificador `UNICODE`. Ahora cuando nosotros hacemos un programa que queremos que use Unicode, lo primero que hacemos es definir el identificador `UNICODE`, como muestra el Listado 34.

```
#define UNICODE
#include <windows.h>
```

```
int main()
{
 MessageBox(0,TEXT("Hola mundo"),TEXT("Mensaje
Unicode"),MB_OK);
 return 0;
}
```

Listado 34: Programa Windows NT que compila tanto en ASCII como en Unicode

El macro `TEXT()` nos devuelve un texto Unicode o ASCII dependiendo de si está definido el identificador `UNICODE`. Todas las funciones de la API de Windows NT tienen el prototipo pensado para aceptar tanto ASCII como Unicode:

```
BOOL SetWindowText(HWND hwnd, LPCTSTR lpText);
```

Y después dependiendo de si está definido o no el identificador `UNICODE` se sustituye la llamada por llamadas a funciones que trabajan con ASCII o con Unicode:

```
#ifdef UNICODE
  #define SetWindowText SetWindowTextW
#else
  #define SetWindowText SetWindowTextA
#endif
```

7.2. Conversiones a otros juegos de caracteres

Si nuestro SO no soporta Unicode, para realizar la conversión al juego de caracteres nativo podemos usar métodos de la clase `String`. En concreto, para convertir un objeto `String` en un array de caracteres nativo recomendamos hacerlo usando:

```
byte[] <String> getBytes()
```

que nos devuelve un array de bytes con los caracteres codificados en el juego de caracteres que usa por defecto la plataforma. También podemos usar:

```
byte[] <String> getBytes(String charset)
```

Para indicar en que juego de caracteres queremos que nos codifique. La siguiente función (que usamos en el aparatado 4.5.3 para hacer los shared stub) es útil para realizar esta conversión:

```
static const char* GetCadenaNativa(JNIEnv *env
                                   , jstring jstr)
{
 char* resultado;
 jbyteArray chars;
 jint longitud;
 static jmethodID MID_String_getBytes;

 // Obtenemos el methodID del String.getBytes()
 MID_String_getBytes=(*env)->GetMethodID(env
        ,(*env)->FindClass(env,"java/lang/String")
                  , "getBytes", "()[B");
```

```
 if (MID_String_getBytes==NULL)
    {
    (*env)->ExceptionDescribe(env);
    return NULL;
    }
 // Lo ejecutamos sobre el String para obtener
 // un array nativo
 chars = (*env)->CallObjectMethod(env,jstr
                                  , MID_String_getBytes);
 if ((*env)->ExceptionOccurred(env))
    {
     (*env)->ExceptionDescribe(env);
     return NULL;
    }
  // Creamos un buffer donde ponemos el array de
 // caracteres obtenido
 longitud = (*env)->GetArrayLength(env,chars);
 resultado = (char*)malloc(longitud+1);
 (*env)->GetByteArrayRegion(env,chars, 0, longitud,
(jbyte*)resultado);
 resultado[longitud] = '\0';

 // Liberamos la referencia local al array de bytes
 (*env)->DeleteLocalRef(env,chars);
 return resultado;
}
```

Para realizar la conversión en el sentido opuesto podemos usar el constructor:

```
String(byte[] bytes)
```

que recibe un array de bytes y lo interpreta según el juego de caracteres de la plataforma nativa. También podemos indicar el juego de caracteres a usar con el constructor:

```
String(byte[] bytes, String charset)
```

7.3. Conversiones a otros juegos de caracteres en el JSDK 1.4

En JSDK 1.4 han surgido nuevas clases para realizar estas conversiones de juegos de caracteres en el paquete `java.nio.charset.*`. Dentro de este paquete encontramos la clase `java.nio.charset.Charset` que sirve para representar los distintos juegos de caracteres. Esta clase es abstracta, con lo que no tiene constructores, sino que para obtener un objeto de este tipo usamos el método estático:

```
static Charset <Charset> forName(String charsetName)
```

Los valores más típicos que podemos pasar en `charserName` se resumen en la Tabla 10.

Juego de caracteres	Descripción
"US-ASCII"	ASCII de 7 bits

"ISO-8859-1"	Juego de caracteres de Mac OS X, Windows o Linux
"MacRoman"	Juego de caracteres de Mac OS Classic
"UTF8"	Juego de caracteres de BeOS

Tabla 10: Juegos de caracteres comunes

Una vez tengamos el objeto `Charset` usamos los métodos:

```
CharsetEncoder <Charset> newEncoder()
CharsetDecoder <Charset> newDecoder()
```

Que nos devuelven unos objetos que podemos usar para codificar o descodificar del charset a Unicode usando:

```
CharBuffer <CharsetDecoder> decode(ByteBuffer in)
ByteBuffer <CharsetEncoder> encode(Charbuffer in)
```

`ByteBuffer` y `CharBuffer` son clases definidas en el paquete `java.nio.*` que actúan como envoltorios alrededor de un conjunto de bytes. El array de bytes puede proceder de distintos canales, típicamente es un `FileChannel`, aunque también podemos crear un `ByteBuffer` a partir de un array de bytes con:

```
static ByteBuffer <ByteBuffer> wrap(byte[] array)
```

P.e. si tenemos un `ByteBuffer` con texto en formato `"ISO-8859-1"` que hemos leído de un fichero y queremos pasarlo a un `String` Java podemos hacer:

```
Charset charset = Charset.forName("ISO-8859-1");
CharsetDecoder decoder = charset.newDecoder();
CharBuffer char_buffer = decoder.decode(buffer);
String texto = char_buffer.toString();
```

Espero que este tutorial le haya resultado útil.

Saludos: Fernando López Hernández

Referencias

[JNIPROG] "The Java Native Interface. Programmer's Guide and Specifications". SHENG LIANG. Ed. Addison Wesley

[JNISPEC] "JNI 1.1 Specification" http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html

[OSXTHREADING] "Mac OS X Threading", http://developer.apple.com/technotes/tn/tn2028.html#MacOSXThreading

[JNILIBRARIES] JNI Libraries http://developer.apple.com/documentation/Java/Conceptual/Java141Development/Core_APIs/chapter_6_section_4.html