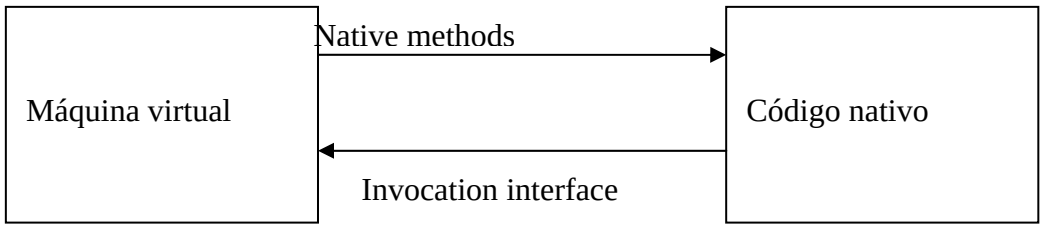
PROGRAMACIÓN C/C++ - JAVA CON JNI

Francisco Javier Rufo Mendo
javirufo@sicubo.com

Índice

Introducción

JNI es un mecanismo que nos permitirá ejecutar código nativo desde Java y viceversa. En este manual nos centraremos en código nativo escrito en C/C++, aunque podrían ser funciones implementadas en otros lenguajes, como Pascal, etc.
Debido a esta bidireccionalidad entre Java y código nativo, tendremos dos tipos de interfaces:
Métodos nativos o *native methods*: Son aquellos que permiten a Java llamar a funciones implementadas en código nativo.
Interfaz de invocación o *invocation interface*: Es aquel que nos permite incrustar una máquina virtual en una aplicación nativa. Por ejemplo, la aplicación *java* es una aplicación nativa que incrusta una máquina virtual.



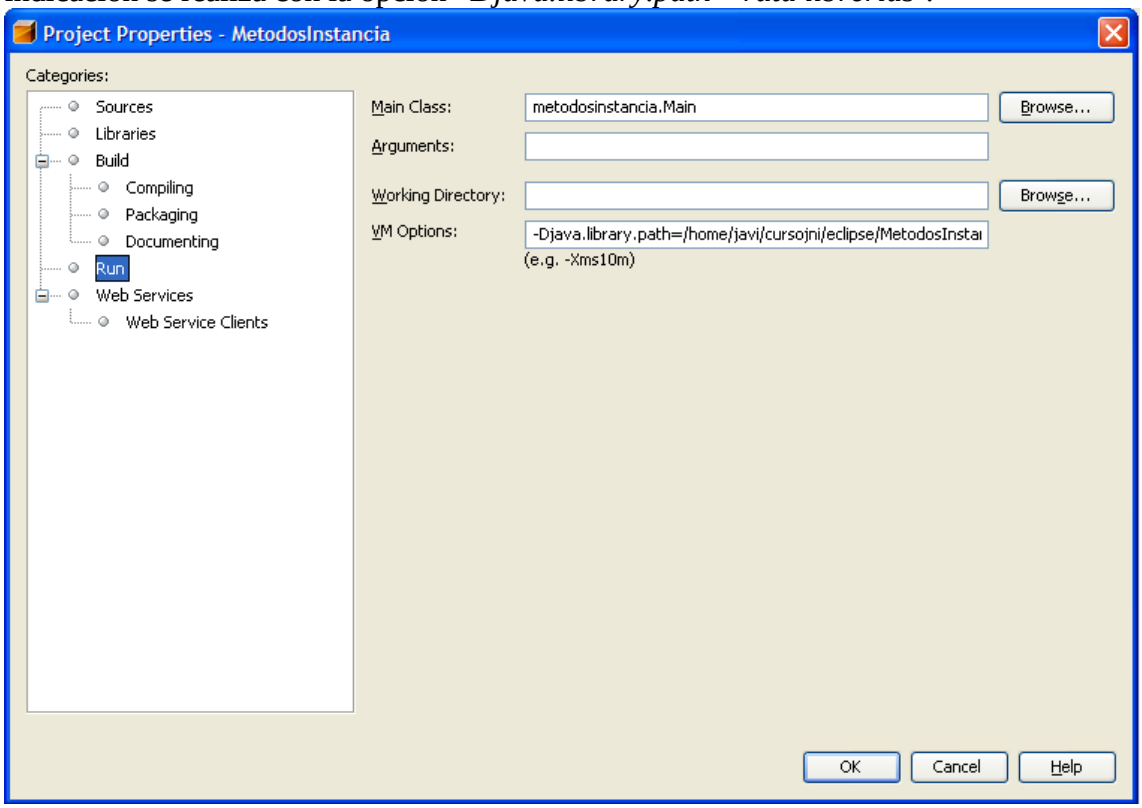
A la hora de llevar a cabo un desarrollo utilizando JNI, habrá que tener en cuenta una seria de consideraciones:
Al utilizar código nativo, las aplicaciones no serán directamente portables entre plataformas, perdiendo una de las características más relevantes de Java.
Java lleva implícito un nivel extra de seguridad en cuando a uso de los tipos, memoria, etc. Esto no sucede en los lenguajes nativos, como C/C++, que es el caso que nos ocupa. Es por ello que el desarrollado deberá tratar de controlar cualquier tipo de error, pues un error en el código nativo podrá desembocar en un mal funcionamiento de la aplicación completa.
Por último, hay que considerar el hecho de que una llamada JNI es más costosa que una llamada Java-Java. Por lo tanto, habrá que reducir el uso de JNI a aquellos en los que se considere imprescindible y realmente sea ventajoso el uso de código nativo frente al uso de código Java.

Configuración del entorno de desarrollo

Para el seguimiento del curso y la implementación de los ejemplos del presente manual nos centraremos en las aplicaciones Netbeans 5.5 y eclipse 3.2. Como máquina virtual Java utilizaremos el JDK 1.5.0_09.
No obstante, el desarrollo puede realizarse con cualquier editor y cualquier máquina virtual Java, si bien, algunas características JNI podrían variar entre versiones.
Por lo tanto, el único requisito para poder llevar a cabo un desarrollo JNI es un JDK y un compilador de C.

Configuración de NetBeans

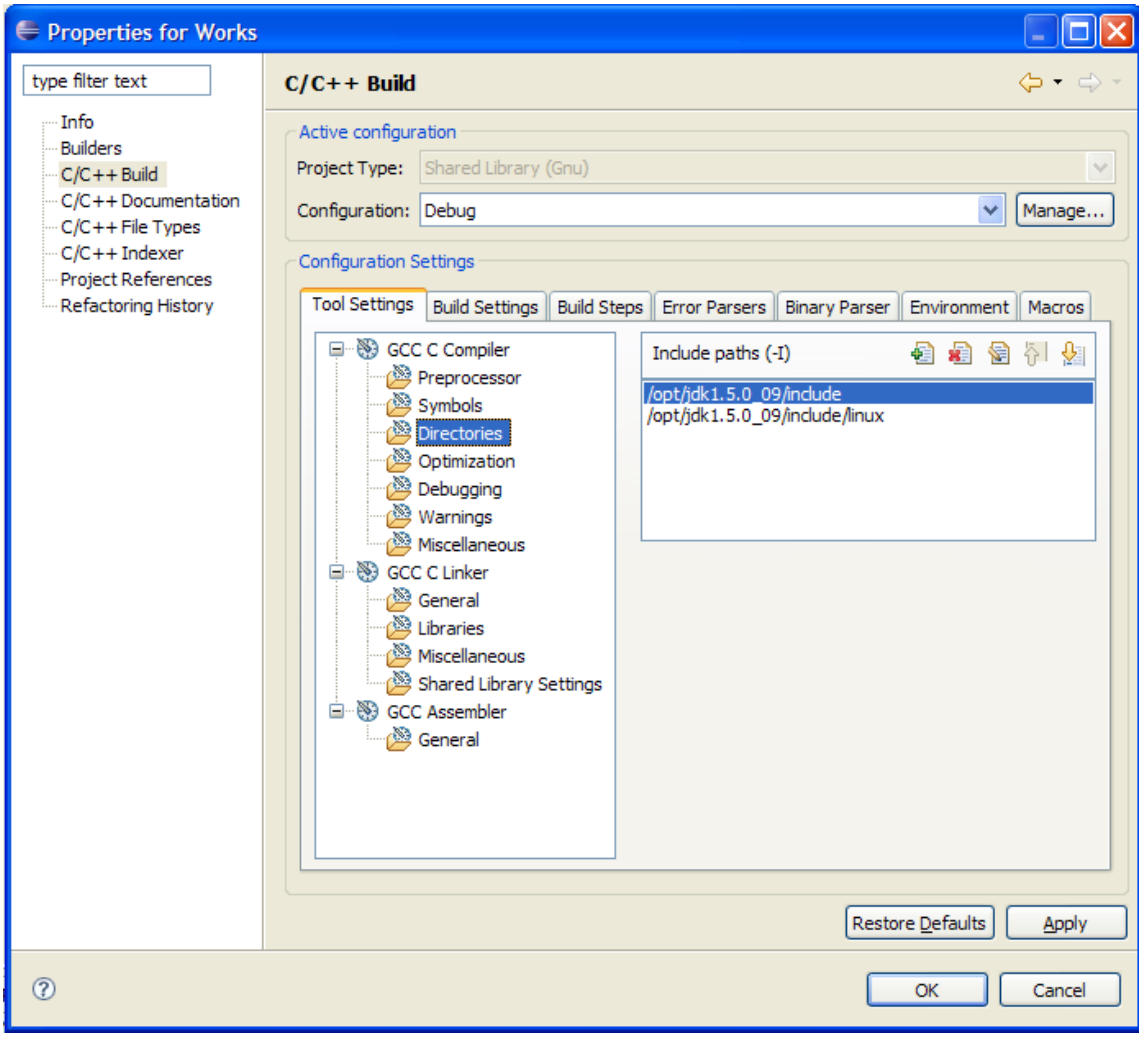
Para generar código que invoque a métodos nativos no se necesita ninguna configuración especial. El único aspecto a tener en cuenta es señalarle al proyecto en sus opciones de configuración la ruta en la que se encuentran las librerías a utilizar. Esta indicación se realiza con la opción *–Djava.library.path='ruta librerias'*.



En esta captura puede verse como se le indica a la aplicación la ruta de las librerías. Esta y otras opciones se establecen en las opciones del proyecto.

Configuración de eclipse

A la hora de trabajar con proyectos en eclipse, siempre crearemos un *Managed Make C Project* y entre los diferentes tipos existentes seleccionaremos *Shared library*.
Tan solo nos quedará incluir en las opciones del proyecto los directorios de *include* adicionales que contienen las definiciones de JNI.



Estos directorios a incluir serán, tomando como base la ubicación del jdk, *incluye* e *include/Linux* o *include/win32*, etc. Según nos encontremos en una u otra plataforma.

HolaMundo con JNI

Para nuestra primera aplicación JNI y ver el proceso de creación de aplicaciones JNI, vamos a llevar a cabo la implementación de un programa Java que invoca a un método nativo JNI. El método nativo se limitará a mostrar un mensaje de texto por consola.

```
package holamundojni;
public class Main {
    public Main() {
    }
    public native void HolaMundoJNI();
    public static void main(String[] args) {
              new Main().HolaMundoJNI();
    }
    static {
        System.loadLibrary("HolaMundoJNI");
    }
}
```

Como puede verse, los métodos nativos se declaran con la palabra reservada *native* y no tienen ningún tipo de implementación. Además, con la invocación de la llamada *System.loadLibrary* indicamos que se cargue la librería que contiene la implementación de los métodos nativos.
El proceso que debe llevarse a cabo es el siguiente:
Compilar las clases mediante el comando *javac*.
Una vez que tenemos las clases compiladas, debemos obtener la definición del fichero cabecera para los métodos JNI. Esto se realiza mediante el comando *javah –jni rutaClase.NombreClase*.

Con esto obtendríamos el fichero de cabeceras *.h*.

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

#ifndef _Included_holamundojni_Main
#define _Included_holamundojni_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     holamundojni_Main
 * Method:    HolaMundoJNI
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_holamundojni_Main_HolaMundoJNI
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Se nos ha generado un fichero con la definición de una única función cuyo nombre es *Java_holamundojni_Main_HolaMundoJNI*. Esta nomenclatura sigue el esquema *Java_rutadelaclase_nombredelaclase_nombremetodonativo*.
También puede verse que aunque el método nativo no tiene parámetros aquí aparecen dos.
JNIEnv: Es un apuntador al entorno JNI. Este entorno contiene datos internos y además un apuntador a una tabla de punteros a funciones JNI.
Jobject: Es una apuntador a la instancia de la clase que ha invocado el método. Si en lugar de un método de instancia invocamos un método de clase, este parámetro será de tipo *jclass*. Se corresponde al puntero *this*.

Nos quedaría por último implementar ese método nativo.

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

JNIEXPORT void JNICALL Java_holamundojni_Main_HolaMundoJNI
  (JNIEnv *env, jobject obj)
  {
      printf("Hola mundo a traves de jni!!!\n");
  }
```

Una vez implementado, tendremos que generar la librería y ejecutar la aplicación Java.

Tipos básicos, arrays y objetos

Tipos básicos

Son tipos cuya correspondencia con tipos de C es directa.

Tipo Java	Tipo C	Descripción
bolean	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
double	jdouble	64 bits formato IEEE
float	jfloat	32 bits formato IEEE

Veremos a continuación un ejemplo de una calculadora que realiza operaciones simples invocando a funciones JNI.

```
package calculadorajni;

import calculo.CalculadoraJNI;
public class Main {
    public Main() {
    }
    public static void main(String[] args) {
        if (args.length!=3)
        {
            System.out.println("Error, solo se admiten tres
parametros");
            return;
        }
        CalculadoraJNI calc = new CalculadoraJNI();
        char operacion = args[1].charAt(0);
        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[2]);
        switch(operacion)
        {
            case '+':
                System.out.println("Suma: "+calc.suma(a, b));
                break;
            case '-':
                System.out.println("Resta: "+calc.resta(a, b));
                break;
            case '*':
                System.out.println("Producto: "+calc.producto(a, b));
                break;
            case '/':
                System.out.println("Division: "+calc.division(a, b));
                break;
            default:
                System.out.println("Operacion no permitida");
        }
    }
}
```

El código de la clase *CalculadoraJNI* es el siguiente.

```
package calculo;
```

```
public class CalculadoraJNI {
    public CalculadoraJNI() {
    }
    //---------------------------------------------------------//
    public static native int suma(int a, int b);
    //---------------------------------------------------------//
    public static native int resta(int a, int b);
    //---------------------------------------------------------//
    public static native int producto(int a, int b);
    //---------------------------------------------------------//
    public static native int division(int a, int b);
    //---------------------------------------------------------//
    static{
        System.loadLibrary("calculo");
    }
}
```

El fichero *.h* correspondiente.

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

#ifndef _CALCULO_H
#define _CALCULO_H
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     calculo_CalculadoraJNI
 * Method:    suma
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_suma
  (JNIEnv *, jclass, jint, jint);

/*
 * Class:     calculo_CalculadoraJNI
 * Method:    resta
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_resta
  (JNIEnv *, jclass, jint, jint);

/*
 * Class:     calculo_CalculadoraJNI
 * Method:    producto
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_producto
  (JNIEnv *, jclass, jint, jint);

/*
 * Class:     calculo_CalculadoraJNI
 * Method:    division
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_division
  (JNIEnv *, jclass, jint, jint);
```

```
#ifdef __cplusplus
}
#endif
#endif
```

Por último, la implementación de dichos métodos.

```
#include "calculo.h"

/*
 * Class:     calculo_CalculadoraJNI
 * Method:    suma
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_suma
  (JNIEnv *env, jclass class, jint a, jint b)
  {
      return a+b;
  }

/*
 * Class:     calculo_CalculadoraJNI
 * Method:    resta
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_resta
(JNIEnv *env, jclass class, jint a, jint b)
  {
      return a-b;
  }


/*
 * Class:     calculo_CalculadoraJNI
 * Method:    producto
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_producto
(JNIEnv *env, jclass class, jint a, jint b)
  {
      return a*b;
  }


/*
 * Class:     calculo_CalculadoraJNI
 * Method:    division
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_calculo_CalculadoraJNI_division
(JNIEnv *env, jclass class, jint a, jint b)
  {
      return a/b;
  }
```

Acceso a cadenas de texto String

El tipo *String* de Java se corresponderá con el tipo *jstring* de C. Este tipo no es directamente reconocible por las rutinas de C, por lo que tendremos que usar un conjunto de funciones para su conversión a un tipo que sea reconocido.

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

Esta función nos retornará un apuntador a la cadena correspondiente a *string* pero reconocida por C. El parámetro *isCopy* nos indicará si el apuntador obtenido es una copia de la cadena Java o por el contrario está apuntando directamente a ella, por lo que no es recomendable modificarlo ya que modificaremos el *String* directamente.

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

Esta función libera el espacio ocupado tras la conversión de un *jstring* a una cadena C.

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

Dado un array de caracteres, esta función crea un nuevo *jstring*.
Por último, tenemos una función que nos indicará la longitud de una cadena dada.

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

Estas funciones nos sirven para convertir a UTF-8. Tenemos un conjunto de funciones con el mismo comportamiento pero para Unicode.

```
const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean*
isCopy)
void RealeaseStringChars(JNIEnv* env, jstring string, const jchar*
chars_buffer)
jsize GetStringLength(JNIEnv* env, jstring string)
jstring NewString(JNIEnv* env, const jchar* ubuffer, jsize length)
```

Ejemplo

Veremos a continuación un ejemplo que llama a un método nativo con un parámetro de tipo *String*. Este método nativo mostrará ese mensaje y además retornará un *String*, el cual será solicitado al usuario por pantalla.

```
package cadenas;
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    private native String tomaTexto(String texto);
    public static void main(String[] args) {
        String texto = new Main().tomaTexto("Inserte un mensaje: ");
        System.out.println("El mensaje es: "+texto);
    }
    static{
        System.loadLibrary("cadenas");
    }
}
```

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

```
#ifndef _CADENAS_H
#define _CADENAS_H
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     cadenas_Main
 * Method:    tomaTexto
 * Signature: (Ljava/lang/String;)Ljava/lang/String;
 */
JNIEXPORT jstring JNICALL Java_cadenas_Main_tomaTexto
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "cadenas.h"

JNIEXPORT jstring JNICALL Java_cadenas_Main_tomaTexto
  (JNIEnv *env, jobject obj,  jstring mensaje)
  {
      char texto[255];
      const jbyte* mensaje_c = (*env)->GetStringUTFChars(env, mensaje,
NULL);
      printf("%s:", mensaje_c);
      (*env)->ReleaseStringUTFChars(env, mensaje, mensaje_c);
      scanf("%s", texto);
      return (*env)->NewStringUTF(env, texto);
  }
```

El esquema de los cuadros de código mostrados es *.java, .h y .c*. Este será el esquema a seguir para el resto de los ejemplos.

Acceso a arrays de tipo básico

Los arrays de tipo básico son aquellos cuyos elementos son de alguno de los tipos básicos enumerados anteriormente. Todos los arays son derivados del tipo base *jarray*.

Tipo Java	Tipo C
[] boolean	jbooleanArray
[]char	jcharArray
[]short	jshortArray
[]int	jintArray
[]long	jlongArray
[]float	jfloatArray
[]double	jdoubleArray

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

Esta función nos retornará la longitud de un array, independientemente de su tipo.
Para cada tipo fundamental existe una función específica que nos convertirá el array a un array reconocido por C.

```
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* GetlntArrayElements(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)
```

De la misma forma, tendremos una función de liberación de ese array C para cada uno de los tipos.

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

Donde mode puede ser 0 si queremos liberar el espacio y copiar el contenido al array Java. JNI_ABORT si queremos liberar, JNI_COMMIT es lo mismo que 0 pero no libera el array.

También disponemos de funciones para la creación de arrays.

```
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 NewlntArray(JNIEnv* env, jsize length)
jlongArray NewLongArray(JNIEnv* env, jsize length)
jfloatArray NewFloatArray(JNIEnv* env, jsize length)
jdoubleArray NewDoubleArray(JNIEnv* env, jsize length)
```

Ejemplo

Vamos a realizar el ejemplo de la calculadora visto anteriormente pero aplicado a vectores.

```
package calculadoravectores;
public class CalculadoraVectores {
    public CalculadoraVectores() {
    }
    public native int[]sumaVectores(int[] a, int []b);
    public native int[]restaVectores(int[] a, int []b);
    public native int[]multiplicaVectores(int[] a, int []b);
    public native int[]divideVectores(int[] a, int []b);
    public final int[]vectorA = {1, 1, 2, 3, 5, 8, 13, 21, 34};
    public final int[]vectorB = {1, 2, 3, 5, 7, 11, 13, 17, 23};
    public static void main(String[] args) {
        if (args.length!=1)
        {
            System.out.printf("Solo se admite el parametro de
operacion");
            return;
        }
```

```
        CalculadoraVectores calc = new CalculadoraVectores();
        int []resultado = null;
        char operacion = args[0].charAt(0);
        switch(operacion)
        {
            case '+':
                resultado = calc.sumaVectores(calc.vectorA,
calc.vectorB);
                break;
            case '-':
                resultado = calc.restaVectores(calc.vectorA,
calc.vectorB);
                break;
            case '*':
                resultado = calc.multiplicaVectores(calc.vectorA,
calc.vectorB);
                break;
            case '/':
                resultado = calc.divideVectores(calc.vectorA,
calc.vectorB);
                break;
            default:
                System.out.println("Operacion no permitida");
        }
        if (resultado!=null)
        {
            for(int i=0; i<resultado.length; i++)
                System.out.print(resultado[i]+" ");
        }
    }

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

}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>

#ifndef CALCULADORAVECTORES_H_
#define CALCULADORAVECTORES_H_
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    sumaVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_sumaVectores
  (JNIEnv *, jobject, jintArray, jintArray);

/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    restaVectores
 * Signature: ([I[I)[I
 */
```

```
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_restaVectores
  (JNIEnv *, jobject, jintArray, jintArray);

/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    multiplicaVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_multiplicaVectores
  (JNIEnv *, jobject, jintArray, jintArray);

/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    divideVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_divideVectores
  (JNIEnv *, jobject, jintArray, jintArray);

#ifdef __cplusplus
}
#endif
#endif /*CALCULADORAVECTORES_H_*/
```

```
#include "calculadoravectores.h"

/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    sumaVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_sumaVectores
  (JNIEnv *env, jobject obj, jintArray vectorA, jintArray vectorB)
{
      jintArray vectorC;
      jint* A;
      jint* B;
      jint* C;
      jsize index;
      jsize longitudA = (*env)->GetArrayLength(env, vectorA);
      jsize longitudB = (*env)->GetArrayLength(env, vectorB);
      if (longitudA != longitudB)
            return NULL;
      vectorC = (*env)->NewIntArray(env, longitudA);
      A = (jint*) (*env)->GetIntArrayElements(env, vectorA, NULL);
      B = (jint*) (*env)->GetIntArrayElements(env, vectorB, NULL);
      C = (jint*) (*env)->GetIntArrayElements(env, vectorC, NULL);
      for(index=0; index<longitudA; index++)
      {
            C[index] = A[index]+B[index];
      }
      (*env)->ReleaseIntArrayElements(env, vectorA, A, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorB, B, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorC, C, 0);//Copio el
contenido y libero C
```

```
      return vectorC;
}
/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    restaVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_restaVectores
  (JNIEnv *env, jobject obj, jintArray vectorA, jintArray vectorB)
  {
      jintArray vectorC;
      jint* A;
      jint* B;
      jint* C;
      jsize index;
      jsize longitudA = (*env)->GetArrayLength(env, vectorA);
      jsize longitudB = (*env)->GetArrayLength(env, vectorB);
      if (longitudA != longitudB)
            return NULL;
      vectorC = (*env)->NewIntArray(env, longitudA);
      A = (jint*) (*env)->GetIntArrayElements(env, vectorA, NULL);
      B = (jint*) (*env)->GetIntArrayElements(env, vectorB, NULL);
      C = (jint*) (*env)->GetIntArrayElements(env, vectorC, NULL);
      for(index=0; index<longitudA; index++)
      {
            C[index] = A[index]-B[index];
      }
      (*env)->ReleaseIntArrayElements(env, vectorA, A, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorB, B, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorC, C, 0);//Copio el
contenido y libero C
      return vectorC;
  }


/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    multiplicaVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_multiplicaVectores
  (JNIEnv *env, jobject obj, jintArray vectorA, jintArray vectorB)
  {
      jintArray vectorC;
      jint* A;
      jint* B;
      jint* C;
      jsize index;
      jsize longitudA = (*env)->GetArrayLength(env, vectorA);
      jsize longitudB = (*env)->GetArrayLength(env, vectorB);
      if (longitudA != longitudB)
            return NULL;
      vectorC = (*env)->NewIntArray(env, longitudA);
      A = (jint*) (*env)->GetIntArrayElements(env, vectorA, NULL);
      B = (jint*) (*env)->GetIntArrayElements(env, vectorB, NULL);
      C = (jint*) (*env)->GetIntArrayElements(env, vectorC, NULL);
      for(index=0; index<longitudA; index++)
      {
```

```
            C[index] = A[index]*B[index];
      }
      (*env)->ReleaseIntArrayElements(env, vectorA, A, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorB, B, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorC, C, 0);//Copio el
contenido y libero C
      return vectorC;
  }

/*
 * Class:     calculadoravectores_CalculadoraVectores
 * Method:    divideVectores
 * Signature: ([I[I)[I
 */
JNIEXPORT jintArray JNICALL
Java_calculadoravectores_CalculadoraVectores_divideVectores
  (JNIEnv *env, jobject obj, jintArray vectorA, jintArray vectorB)
  {
      jintArray vectorC;
      jint* A;
      jint* B;
      jint* C;
      jsize index;
      jsize longitudA = (*env)->GetArrayLength(env, vectorA);
      jsize longitudB = (*env)->GetArrayLength(env, vectorB);
      if (longitudA != longitudB)
            return NULL;
      vectorC = (*env)->NewIntArray(env, longitudA);
      A = (jint*) (*env)->GetIntArrayElements(env, vectorA, NULL);
      B = (jint*) (*env)->GetIntArrayElements(env, vectorB, NULL);
      C = (jint*) (*env)->GetIntArrayElements(env, vectorC, NULL);
      for(index=0; index<longitudA; index++)
      {
            C[index] = A[index]/B[index];
      }
      (*env)->ReleaseIntArrayElements(env, vectorA, A, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorB, B, JNI_ABORT);
      (*env)->ReleaseIntArrayElements(env, vectorC, C, 0);//Copio el
contenido y libero C
      return vectorC;
  }
```

Acceso a arrays de referencias

Los arrays de referencias son aquellos del tipo *jobjectArray*. A diferencia de los arrays de tipos básicos, los elementos de estos arrays deben ser accedidos elemento a elemento.

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

Estos métodos permiten obtener un elemento de array dada su posición y estableces un elemento del array dada su posición y nuevo valor.
Al igual que con los arrays básicos, también tenemos una función para crear un nuevo array.

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

En este caso, deberemos también pasarle un valor al que inicializar los elementos del array, pudiendo ser NULL.
Para obtener el *jclass* correspondiente a la clase de los elementos del array se usa la siguiente función.

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

Cuyo funcionamiento vamos a ver más adelante.

Ejemplo

Vamos a seguir con el ejemplo de la calculadora ampliándolo al uso de matrices. En este caso tan sólo veremos la operación de suma de matrices. Para obtener el *jclass* del método de creación de arrays, indicaremos que vamos a utilizar la clase *"[I"*.

```
package calculadoramatrices;
public class CalculadoraMatrices {

    public CalculadoraMatrices() {
    }
    public native int[][]sumaMatrices(int [][]a, int [][]b);
    public static void main(String[] args) {
        int [][]a = {{1,0,3},{1,2,3},{1,7,3}};
        int [][]b = {{8,2,3},{1,5,3},{1,2,3}};
        int [][]c = new CalculadoraMatrices().sumaMatrices(a, b);
        if (c!=null)
        {
            for(int i=0; i<c.length; i++)
            {
                for(int j=0; j<c[0].length; j++)
                {
                    System.out.print(c[i][j]+" ");
                }
                System.out.println();
            }
        }
    }
    static{
        System.loadLibrary("CalculadoraMatrices");
    }
}
```

```
#include <jni.h>
#ifndef CALCULADORAMATRICES_H_
#define CALCULADORAMATRICES_H_
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     calculadoramatrices_CalculadoraMatrices
 * Method:    sumaMatrices
 * Signature: ([[I[[I)[[I
 */
JNIEXPORT jobjectArray JNICALL
Java_calculadoramatrices_CalculadoraMatrices_sumaMatrices
```

```
  (JNIEnv *, jobject, jobjectArray, jobjectArray);

#ifdef __cplusplus
}
#endif
#endif /*CALCULADORAMATRICES_H_*/
```

```
#include "calculadoramatrices.h"
JNIEXPORT jobjectArray JNICALL
Java_calculadoramatrices_CalculadoraMatrices_sumaMatrices
  (JNIEnv *env, jobject obj, jobjectArray matrizA, jobjectArray
matrizB)
  {
            jobjectArray matrizC;
            jsize index;
            jclass tipoMatriz;
            jsize filas = (*env)->GetArrayLength(env, matrizA);
            if (filas!=(*env)->GetArrayLength(env, matrizB))
                  return NULL;
            tipoMatriz = (*env)->FindClass(env, "[I");
            if(tipoMatriz==NULL)
                  return NULL;
            matrizC = (*env)->NewObjectArray(env, filas, tipoMatriz,
NULL);
            if(matrizC==NULL)
                  return NULL;
            printf("Tengo %d filas\n", filas);
            for(index=0; index<filas; index++)
            {
                  printf("Proceso fila %d\n", index);
                  jsize col;
                  jint *A;
                  jint *B;
                  jint *C;
                  jintArray filaA = (jintArray) (*env)-
>GetObjectArrayElement(env, matrizA, index);
                  jintArray filaB = (jintArray) (*env)-
>GetObjectArrayElement(env, matrizB, index);
                  jsize columnas = (*env)->GetArrayLength(env, filaA);

                  jintArray filaC = (*env)->NewIntArray(env, columnas);

                  A = (*env)->GetIntArrayElements(env, filaA, NULL);

                  B = (*env)->GetIntArrayElements(env, filaB, NULL);

                  C = (*env)->GetIntArrayElements(env, filaC, NULL);

                  for(col=0; col<columnas; col++)
                  {
                        printf("%d + %d\n", A[col], B[col]);
                        C[col] = A[col]+B[col];
                  }

                  (*env)->ReleaseIntArrayElements(env, filaA, A,
JNI_ABORT);
                  (*env)->ReleaseIntArrayElements(env, filaB, B,
JNI_ABORT);
                  (*env)->ReleaseIntArrayElements(env, filaC, C, 0);
```

```
                    (*env)->SetObjectArrayElement(env, matrizC, index,
filaC);
                    (*env)->DeleteLocalRef(env, filaA);
                    (*env)->DeleteLocalRef(env, filaB);
                    (*env)->DeleteLocalRef(env, filaC);
            }

            return matrizC;
  }
```

Acceso a objetos

Signatura de tipo

Para acceder los atributos y métodos de Java mediante JNI, necesitamos especificar los tipos de los datos. Para ello utilizamos las signaturas de tipo o *member description.* La signatura de los tipos básicos se representan mediante letras.

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

Si deseamos indicar un array de elementos, el tipo se precede por *"[*", quedando por ejemplo, para un array de enteros *"[I"*. Si desearamos una matris de *int "[[I"*.
Para las clases, la forma de indicar la signatura es *L* seguido de la ruta de la clase utilizando *'/'* como separador en lugar del punto. Por último se especifica el nombre y se finaliza con *';'*. Por ejemplo, la signatura de la clase *String* sería *"Ljava/lang/String;"*.

Cuando especificamos la signatura de un método, si este recibe varios parámetros, se pondrán todas las signaturas seguidas sin ningún tipo de separador.
Cuando un método no tiene ningún valor de retorno, deberá especificarse en la signatura mediante *V*. Por el contrario, si no tiene ningún parámetro, no habrá que indicar nada.
A continuación se muestran algunos ejemplos.
int metodo(int a, int b) ➔ (II)I
void metodo() ➔ ()V
int metodo(int []a, String b) ➔ ([ILjava/lang/String;)I

Si deseamos obtener la signatura de lo métodos para una clase dada, puede utilizarse el comando *"javap –s"*.

Acceso atributos de objeto

Acceso atributos de instancia

Para acceder un atributo de instancia, deberemos obtener primero el identificador del atributo a acceder y después podremos recuperar su valor o modificarlo.

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

Este método nos permitirá obtener el ID de un campo, indicando su nombre y signatura. El parámetro *class* podremos obtenerlo mediante el uso de la función *GetObjectClass*,

que recibe por parámetro el apuntador al entorno y el objecto del cual queremos obtener la clase.
Por último, tenemos diferentes métodos para acceder o modificar ese campo según sea su tipo. Recuérdese que los elementos cuyo tipo sea una clase se tratarán como de tipo *Object*.

```
jboolean GetBooleanField(JNIEnv* env, jobject object, jfieldlD
fieldlD)
jbyte GetByteField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jshort GetShortField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jchar GetCharField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jint GetlntField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jlong GetLongField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jfloat GetFloatField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jdouble GetDoubleField(JNIEnv* env, jobject object, jfieldlD fieldlD)
jobject GetObjectField(JNIEnv* env, jobject object, jfieldlD fieldlD)
```

Estas funciones nos permiten recuperar el valor del campo.

```
void SetBooleanField(JNIEnv* env, jobject obj, jfieldlD fieldlD,
jboolean value)
void SetByteField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jbyte
value)
void SetShortField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jshort
value)
void SetCharField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jchar
value)
void SetlntField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jint
value)
void SetLongField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jlong
value)
void SetFloatField(JNIEnv* env, jobject obj, jfieldlD fieldlD, jfloat
value)
void SetDoubleField(JNIEnv* env, jobject obj, jfieldlD fieldlD,
jdouble value)
void SetObjectField(JNIEnv* env, jobject obj, jfieldlD fieldlD,
jobject value)
```

Estas funciones nos permiten modificar el valor del campo.

Ejemplo

Programa que accede un atributo de tipo String de un objeto y lo modifica.

```
package cadenas2;
public class Main {

    private String cadena;
    public Main(String texto) {
        cadena = new String(texto);
    }

    public native void ModificaCadena();
    public static void main(String[] args) {
        // TODO code application logic here

        Main cadenas2 = new Main("Esto es una cadena");
        System.out.println("La cadena de Main es: "+cadenas2.cadena);
        cadenas2.ModificaCadena();
```

```
        System.out.println("La cadena de Main es: "+cadenas2.cadena);
    }

    static {
        System.loadLibrary("cadenas2");
    }
}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class cadenas2_Main */

#ifndef _Included_cadenas2_Main
#define _Included_cadenas2_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     cadenas2_Main
 * Method:    ModificaCadena
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_cadenas2_Main_ModificaCadena
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "cadenas2.h"
#include <string.h>
#include <malloc.h>

JNIEXPORT void JNICALL Java_cadenas2_Main_ModificaCadena
  (JNIEnv *env, jobject obj)
  {

      jclass clase = (*env)->GetObjectClass(env, obj);
      if (clase==NULL)
      {
            printf("No se encuentra clase\n");
            return;
      }
      jfieldID campoID = (*env)->GetFieldID(env, clase, "cadena",
"Ljava/lang/String;");
      if (campoID==NULL)
      {
            printf("No se encuentra el campo\n");
            return;
      }
      jstring cadena = (*env)->GetObjectField(env, obj, campoID);
      const jbyte* cadenaC = (*env)->GetStringUTFChars(env, cadena,
NULL);
```

```
      (*env)->SetObjectField(env, obj, campoID, (*env)-
>NewStringUTF(env,  strcat(cadenaC, "HOLA")));
      (*env)->ReleaseStringUTFChars(env, cadena, cadenaC);
  }
```

Acceso atributos de clase

Los atributos de clase son aquellos que están declarados como *static*. El mecanismo es el mismo que para los atributos de instancia, salvo que se utilizan otros métodos.

```
jfiledlD GetStaticFieldlD(JNIEnv* env, jclass class, cost char* name,
const char* signature)
```

En este caso para obtener el *class* se utiliza la función *FindClass*, que recibe por parámetro el apuntador al entorno y el nombre de la clase.

```
jboolean GetStaticBooleanF±led(JNIEnv* env, jclass class, jfieldlD
fieldlD)
jbyte GetStaticByteFiled(JNIEnv* env, jclass class, jfieldlD fieldlD)
jshort GetStaticShortFiled(JNIEnv* env, jclass class, jfieldlD
fieldlD)
jchar GetStaticCharFiled(JNIEnv* env, jclass class, jfieldlD fieldlD)
jint GetStaticlntFiled(JNIEnv* env, jclass class, jfieldlD fieldlD)
jlong GetStaticLongFiled(JNIEnv* env, jclass class, jfieldlD fieldlD)
jfloat GetStaticFloatFiled(JNIEnv* env, jclass class, jfieldlD
fieldlD)
jdouble GetStaticDoubleFiled(JNIEnv* env, jclass class, jfieldlD
fieldlD)
jobject GetStaticObjectFiled(JNIEnv* env, jclass class, jfieldlD
fieldlD)
```

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

Ejemplo

```
package cadenas2;

public class Main {

    private static String cadena;
```

```
    public Main(String texto) {
        cadena = new String(texto);
    }

    public native void ModificaCadena();
    public static void main(String[] args) {
        // TODO code application logic here

        Main cadenas2 = new Main("Esto es una cadena");
        for(int i=0; i<100; i++)
        {
            System.out.println("La cadena de Main es:
"+cadenas2.cadena);
            cadenas2.ModificaCadena();
        }
        System.out.println("La cadena de Main es: "+cadenas2.cadena);
    }

    static {
        System.loadLibrary("cadenas2");
    }
}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class cadenas2_Main */

#ifndef _Included_cadenas2_Main
#define _Included_cadenas2_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     cadenas2_Main
 * Method:    ModificaCadena
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_cadenas2_Main_ModificaCadena
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "cadenas2.h"
#include <string.h>
#include <malloc.h>

JNIEXPORT void JNICALL Java_cadenas2_Main_ModificaCadena
  (JNIEnv *env, jobject obj)
  {

      jclass clase = (*env)->GetObjectClass(env, obj);
      if (clase==NULL)
      {
            printf("No se encuentra clase\n");
            return;
      }
```

```
      jfieldID campoID = (*env)->GetStaticFieldID(env, clase,
"cadena", "Ljava/lang/String;");
      if (campoID==NULL)
      {
            printf("No se encuentra el campo\n");
            return;
      }
      jstring cadena = (*env)->GetStaticObjectField(env, obj,
campoID);
      const jbyte* cadenaC = (*env)->GetStringUTFChars(env, cadena,
NULL);
      jstring nuevaCadena = (*env)->NewStringUTF(env,
"HOLA");//strcat(cadenaC, "HOLA"));
      (*env)->SetStaticObjectField(env, obj, campoID, nuevaCadena);
      (*env)->ReleaseStringUTFChars(env, cadena, cadenaC);

  }
```

Acceso a métodos de un objeto

El acceso a un método Java se conoce también como *callback*. Esto es debido a que Java invoca un método nativo que a su vez invoca uno o varios métodos Java.

Acceso a métodos de instancia

Al igual que sucede con los atributos, para poder acceder un método deberemos obtener su ID.

```
jmethodlD GetMethodlD(JNIEnv* env, jclass class, const char* name,
const char* signature)
```

Una vez obtenido el ID, podremos invocarlo. Se utiliza una función de invocación en base al tipo del retorno.

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

La lista de parámetros variables al final se utiliza para indicar los parámetros que recibirá el método.

Ejemplo

Vamos a implementar un método nativo que invoca a un método Java que suma dos números.

```
package metodosinstancia;
public class Main {
    private int resultado;
    public Main() {
    }

    public int getResultado(){
        return resultado;
    }
    public void suma(int a, int b){
        resultado = a + b;
    }
    public native void sumaC();
    public static void main(String[] args) {
        Main metodos = new Main();
        System.out.println("Antes de JNI resultado es: "
+metodos.getResultado());
        metodos.sumaC();
        System.out.println("Despues de JNI resultado es: "
+metodos.getResultado());
    }
    static {
        System.loadLibrary("MetodosInstancia");
    }
}
```

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

#ifndef _Included_metodosinstancia_Main
#define _Included_metodosinstancia_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodosinstancia_Main
 * Method:    sumaC
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_metodosinstancia_Main_sumaC
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "metodos.h"

JNIEXPORT void JNICALL Java_metodosinstancia_Main_sumaC
  (JNIEnv *env,  jobject obj)
  {
      jclass clase = (*env)->FindClass(env, "metodosinstancia/Main");
```

```
      if (clase==NULL)
            return;
      jmethodID metodoID = (*env)->GetMethodID(env, clase, "suma",
"(II)V");
      if (metodoID==NULL)
            return;
      (*env)->CallVoidMethod(env, obj, metodoID, 10, 15);
  }
```

Acceso a métodos de clase

El mecanismo es idéntico a los métodos de instancia, pero en este caso utilizaremos las funciones que se indican a continuación.

```
jmethodlD GetStaticMethodlD(JNIEnv* env, jclass class, const char*
name, const char* signature)
void CallStaticVoidMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ...)
jbyte CallStaticByteMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ...)
jshort CallStaticShortMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jchar CallStaticCharMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jint CallStaticlntMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jlong CallStaticLongMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jfloat CallStaticFloatMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jdouble CallStaticDoubleMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
jobject CallStaticObjectMethod(JNIEnv* env, jclass class, jmethodlD
methodlD, ..)
```

Ejemplo

Aplicación que invoca a un método nativo. En este caso, el método nativo llama a un método de clase que imprime un mensaje que recibe por parámetro.

```
package metodosclase;

public class Main {

    public Main() {
    }

    public static void Imprime(String mensaje){
        System.out.println(mensaje);
    }
    public native void LlamaImprime();
    public static void main(String[] args) {
        new Main().LlamaImprime();
    }
    static{
        System.loadLibrary("MetodosClase");
```

```
    }
}
```

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

#ifndef _Included_metodosclase_Main
#define _Included_metodosclase_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodosclase_Main
 * Method:    LlamaImprime
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_metodosclase_Main_LlamaImprime
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

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

JNIEXPORT void JNICALL Java_metodosclase_Main_LlamaImprime
  (JNIEnv * env, jobject obj)
  {
      jclass clase = (*env)->FindClass(env, "metodosclase/Main");
      if (clase==NULL)
            return;
      jmethodID metodoID = (*env)->GetStaticMethodID(env, clase,
"Imprime", "(Ljava/lang/String;)V");
      if (metodoID == NULL)
            return;
      jstring cadenaJava = (*env)->NewStringUTF(env, "Hola caracola");
      (*env)->CallStaticVoidMethod(env, obj, metodoID, cadenaJava);
  }
```

Ejecutar métodos de instancia de la superclase

Desde JNI, podemos obtener la clase base de una clase dada con la siguiente función.

```
jclass GetSuperClass(JNIEnv* env, jclass class)
```

Si utilizamos las funciones vistas para ejecución de métodos de una clase, se ejecutarán los métodos de la clase derivada. Si lo que se desea es ejecutar un método de la clase base, deberemos utilizar las funciones de invocación de métodos *NonVirtual*.

```
void CallNonvirtualVoidMethod(JNIEnv* env, jobject obj, jclass class,
jmethodlD methodlD, ...)
jbyte CallNonvirtualByteMethod(JNIEnv* env, jobject obj, jclass class,
jmethodlD methodlD, ...)
jshort CallNonvirtualShortMethod(JNIEnv* env, jobject obj, jclass
class, jmethodlD methodlD, ...)
```

```
jchar CallNonvirtualCharMethod(JNIEnv* env, jobject obj, jclass class,
jmethodlD methodlD, ...)
jint CallNonvirtuallntMethod(JNIEnv* env, jobject obj, jclass class,
jmethodlD methodlD, ...)
jlong CallNonvirtualLongMethod(JNIEnv* env, jobject obj, jclass class,
jmethodlD methodlD, ...)
jfloat CallNonvirtualFloatMethod(JNIEnv* env, jobject obj, jclass
class, jmethodlD methodlD, ...)
jdouble CallNonvirtualDoubleMethod (JNIEnv* env, jobject obj, jclass
class, jmethodlD methodlD,..)
jobject CallNonvirtualObjectMethod(JNIEnv* env, jobject obj, jclass
class, jmethodlD methodlD,..)
```

Ejemplo

Tenemos una clase base con un método que imprime un mensaje. A su vez tenemos otra clase que hereda de la base y tiene el método sobrecargado. Mediante código JNI invocaremos ambos métodos.

Clase base

```
package metodossuperclase;
public class ClaseBase {

    /** Creates a new instance of ClaseBase */
    public ClaseBase() {
    }
    public void Imprime(){
        System.out.println("Estoy en la clase base");
    }
    public native void metodoNativo();

    static{
        System.loadLibrary("MetodosSuperClase");
    }
}
```

Clase derivada

```
package metodossuperclase;
public class Main extends ClaseBase {
    public Main() {
    }


    public void Imprime(){
        System.out.println("Estoy en la clase derivada");
    }
    public static void main(String[] args) {
        ClaseBase base = new Main();
        base.metodoNativo();
    }

}
```

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
```

```
/* Header for class metodossuperclase_ClaseBase */

#ifndef _Included_metodossuperclase_ClaseBase
#define _Included_metodossuperclase_ClaseBase
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodossuperclase_ClaseBase
 * Method:    metodoNativo
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_metodossuperclase_ClaseBase_metodoNativo
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "metodos.h"
JNIEXPORT void JNICALL Java_metodossuperclase_ClaseBase_metodoNativo
  (JNIEnv *env, jobject obj)
  {
      jclass clase = (*env)->FindClass(env,
"metodossuperclase/ClaseBase");
      if (clase==NULL)
      {
            printf("No se encuentra la clase\n");
            return;
      }
      jmethodID metodoID = (*env)->GetMethodID(env, clase, "Imprime",
"()V");
      if (metodoID==NULL)
      {
            printf("No se encuentra el metodo\n");
            return;
      }
      (*env)->CallVoidMethod(env, obj, metodoID);
      (*env)->CallNonvirtualVoidMethod(env, obj, clase, metodoID);
  }
```

Invocar a constructores

La invocación de los constructores se realiza de una forma idéntica a los métodos normales, tendiendo en cuenta que en este caso los métodos reciben el nombre *"<init>"*.

Si lo que deseamos es crear una nueva instancia de una determinada clase, utilizaremos la función *NewObject*.

```
jobject NewObject(JNIEnv, jclass class, jmethodlD constructorlD, ..)
```

Ejemplo

Función nativa que crea dos objetos de una clase determinada e invoca un método Java.

package metodosconstructores;

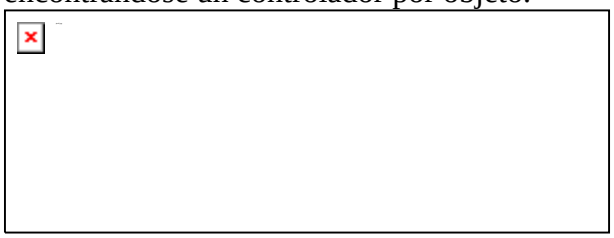
```
public class Main {
    private String _cadena;
    private int _a, _b;
    /** Creates a new instance of Main */
    public Main() {
        _cadena = new String();
        _a = -1;
        _b = -1;
    }

    public Main(String cadena, int a, int b){
        _cadena = new String(cadena);
        _a = a;
        _b = b;
    }

    public void ImprimeValores(){
        System.out.println("Valor de cadena: "+_cadena);
        System.out.println("Valor de a: "+_a);
        System.out.println("Valor de b: "+_b);
    }


    public static native void Constructores();
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        // TODO code application logic here
        Main.Constructores();
    }
    static{
        System.loadLibrary("MetodosConstructores");
    }
}
```

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

#ifndef _Included_metodosconstructores_Main
#define _Included_metodosconstructores_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodosconstructores_Main
 * Method:    Constructores
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_metodosconstructores_Main_Constructores
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "metodos.h"

JNIEXPORT void JNICALL Java_metodosconstructores_Main_Constructores
  (JNIEnv *env, jclass clase)
  {
      jmethodID constructorSimple = (*env)->GetMethodID(env, clase,
"<init>", "()V");
      if(constructorSimple==NULL)
      {
            printf("Metodo no encontrado\n");
            return;
      }
      printf("Objeto1 a crear\n");
      jobject objeto1 = (*env)->NewObject(env, clase,
constructorSimple);
      printf("Objeto1 creado\n");
      jmethodID constructorComplejo = (*env)->GetMethodID(env, clase,
"<init>", "(Ljava/lang/String;II)V");
      if(constructorComplejo==NULL)
      {
            printf("Metodo no encontrado\n");
            return;
      }
      printf("Objeto2 a crear\n");
      jobject objeto2 = (*env)->NewObject(env, clase,
constructorComplejo, (*env)->NewStringUTF(env, "HOLA"), 1, 2);
      printf("Objeto2 creado\n");
      jmethodID Imprime = (*env)->GetMethodID(env, clase,
"ImprimeValores", "()V");
      if(Imprime==NULL)
      {
            printf("Metodo no encontrado\n");
            return;
      }
      (*env)->CallVoidMethod(env, objeto1, Imprime);
      (*env)->CallVoidMethod(env, objeto2, Imprime);
  }
```

Referencias locales y globales

Las referencias JNI son elementos del tipo *jobject*, *jclass*, *jarray* o *jstring*.
Es importante diferenciar entre referencia JNI y objecto Java. Una referencia es una variable de los tipos indicados, que apunta a una estructura con información sobre el objeto Java. El objeto Java es un objeto correspondiente a la máquina virtual al cual accedemos mediante la referencia.
A esa estructura opaca a la que accedemos se la denomina *controlador de objeto*, encontrándose un controlador por objeto.



Hay tres tipos de referencias:
Locales
Globales
Globales desligadas

Referencias locales

Una referencia local es una referencia que es válida sólo durante la llamada a un método nativo. Estas referencias son liberadas por INI una vez acaba la llamada al método nativo.
Además las referencias locales son válidas sólo dentro del hilo que la creo, con lo que una referencia creada en un hilo no debe pasarse a otro hilo. Hay dos formas de eliminar una referencia:
Al acabar la llamada al método nativo se libera la referencia automáticamente
Usar la función JNI *DeleteLocalRef*.

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

Normalmente no liberaremos las referencias y esperaremos a que finalice la llamada JNI para que se liberen automáticamente. Hay varios casos en los que sí se recomienda liberar las referencias de forma manual.
Cuando se necesitan crear un gran número de referencias en una sola función nativa.
Cuando se invoca a funciones auxiliares desde una función JNI.
Cuando el método nativo no retorna, sino que entra en un bucle infinito.

Refencias globales

Son aquellas que sobreviven a la terminación de un método nativo, por lo que podemos seguir accediendo a ellas en sucesivas ocasiones.

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

Nos permite obtener una referencia global a partir de una referencia local ya existente. En estas referencias globales no hay recogida de basura, por lo que debemos liberarla manualmente para que se produzca.

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

Ejemplo

Uso de una referencia global.

```
package referenciaglobal;
public class Main {
    public Main() {
    }

    public static void Imprime(String mensaje){
        System.out.println(mensaje);
    }
    public native void LlamaImprime();
    public static void main(String[] args) {
        // TODO code application logic here
        Main m = new Main();

        for(int i=0; i<10; i++)
           m.LlamaImprime();
    }
    static{
        System.loadLibrary("ReferenciaGlobal");
    }
}
```

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

#ifndef _Included_metodosclase_Main
#define _Included_metodosclase_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodosclase_Main
 * Method:    LlamaImprime
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_referenciaglobal_Main_LlamaImprime
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

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

JNIEXPORT void JNICALL Java_referenciaglobal_Main_LlamaImprime
  (JNIEnv * env, jobject obj)
  {
```

```
      static jclass clase = NULL;
      if (clase==NULL)
      {
            clase = (*env)->FindClass(env, "referenciaglobal/Main");
            if (clase==NULL)
            {
                  printf("No encuentro la clase");
                  return;
            }
            clase = (jclass) (*env)->NewGlobalRef(env, clase);
      }
      else
      {
            printf("Uso la clase ya cargada estatica\n");
      }
      jmethodID metodoID = (*env)->GetStaticMethodID(env, clase,
"Imprime", "(Ljava/lang/String;)V");
      if (metodoID == NULL)
            return;
      jstring cadenaJava = (*env)->NewStringUTF(env, "Hola caracola");
      (*env)->CallStaticVoidMethod(env, obj, metodoID, cadenaJava);
  }
```

Referencias globales desligadas

También son conocidas como *weak global references*. Son idénticas a las referencias globales, pero en este tipo no se garantiza que no se ejecute el recolector de basura sobre ellas. Es por esto que cada vez que se intente hacer uso de una de estas referencias habrá que comprobar que es válida.

```
jweak NewWeakGlobalRef(JNIEnv* env, jobject obj)
void DeleteWeakGlobalRef(JNIEnv* env, jweak wref)
```

Mediante el uso de estas funciones podremos crear y liberar referencias desligadas.
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.

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 IsSameObject.

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

La función devuelve JNI TRUE o JNI FALSE.
Con esta función podremos comprobar si una referencia desligada sigue siendo válida o no mediante el código que se muestra a continuación.

```
if ( ! (*env)_>IsSameobject(env,miobjdesligado,NULL)
{
//Codigo referencia valida
}
else
{
//Codigo referencia no valida
```

```
}
```

Excepciones

Vamos a ver ahora el manejo de excepciones desde JNI. A diferencia de Java, cuando se produce una excepción en JNI, el flujo de ejecución de la aplicación no se detiene. Es por esto que es un punto muy importante el control de excepciones dentro del código JNI.

Captura de excepcionas en JNI

Existe un flag por cada hilo que indica si se ha producido una excepción o no. Tenemos dos funciones que nos permiten comprobar si se ha producido una excepción o no.

```
jboolean ExceptionCheck (JNIEnv* env)
jthrowable ExceptionOccurred(JNIEnv* env)
```

La primera de ellas, nos indica si ha habido o no excepción. Por otro lado, la segunda nos indica si ha habido excepción retornándonos una referencia a la expcepción ocurrida o NULL en caso de no haber ocurrido.
No es seguro ejeutar código tras aparecer una excepción, pues el comportamiento puede ser inesperado. Hay un conjunto de funciones cuya ejecución si es segura.

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

Estas funciones nos permiten eliminar el flag de excepción y liberar recursos antes de retornar a Java.
Una vez que se produce la excepción, esta quedará activa hasta que se de uno de los siguientes casos:
Se desactiva el flag de excepción con *ExceptionClear*.

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

Se retorna del método nativo, por lo cual la excepción se lanza a Java.

También es posible imprimir la pila de llamadas que ha producido la excepción. Esto se hace con la función *ExceptionDescribe*.

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

Ejemplo

Ejemplo de excepciones en JNI. Tendremos 3 funciones, cada una de ellas provocará una excepción. La primera no hará nada y la excepción llegará a Java. La segunda simplemente la limpiará. La tercera, la describirá y la limpiará.

```
package capturaexcepciones;
public class Main {
    public Main() {
```

```
    }

    public static native void noCapturaExcepcion();
    public static native void clearExcepcion();
    public static native void describeExcepcion();

    public static void main(String[] args) {
        System.out.println("****NADA****");
        try{
            Main.noCapturaExcepcion();
        }catch(java.lang.NoClassDefFoundError ex)
        {
            System.out.println("Excepcion capturada en Java:
"+ex.getMessage());
        }        //Sin capturar excepcion, la capturo en Java
        //Clear excepcion
        System.out.println("****CLEAR****");
        Main.clearExcepcion();
        //Describe

        System.out.println("****DESCRIBE****");
        Main.describeExcepcion();
    }

    static{
        System.loadLibrary("CapturaExcepciones");
    }
}
```

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

#ifndef _Included_capturaexcepciones_Main
#define _Included_capturaexcepciones_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     capturaexcepciones_Main
 * Method:    noCapturaExcepcion
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_noCapturaExcepcion
  (JNIEnv *, jclass);

/*
 * Class:     capturaexcepciones_Main
 * Method:    clearExcepcion
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_clearExcepcion
  (JNIEnv *, jclass);

/*
 * Class:     capturaexcepciones_Main
 * Method:    describeExcepcion
 * Signature: ()V
 */
```

```
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_describeExcepcion
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "captura.h"
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_noCapturaExcepcion
  (JNIEnv *env, jclass class)
  {
      (*env)->FindClass(env, "paquito/chocolatero");
      if ((*env)->ExceptionOccurred(env))
      {
            printf("Ha ocurrido una excepcion en JNI. No hacemos
nada\n");
      }
  }

/*
 * Class:     capturaexcepciones_Main
 * Method:    clearExcepcion
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_clearExcepcion
  (JNIEnv *env, jclass class)
  {
      (*env)->FindClass(env, "paquito/chocolatero");
      if ((*env)->ExceptionOccurred(env))
      {
            printf("Ha ocurrido una excepcion en JNI. Vamos a
clear\n");
            (*env)->ExceptionClear(env);
      }
  }


/*
 * Class:     capturaexcepciones_Main
 * Method:    describeExcepcion
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_capturaexcepciones_Main_describeExcepcion
  (JNIEnv *env, jclass class)
  {
      (*env)->FindClass(env, "paquito/chocolatero");
      if ((*env)->ExceptionOccurred(env))
      {
            printf("Ha ocurrido una excepcion en JNI. Vamos a
describe:\n");
            (*env)->ExceptionDescribe(env);
            (*env)->ExceptionClear(env);
      }
  }
```

Lanzar excepciones desde JNI

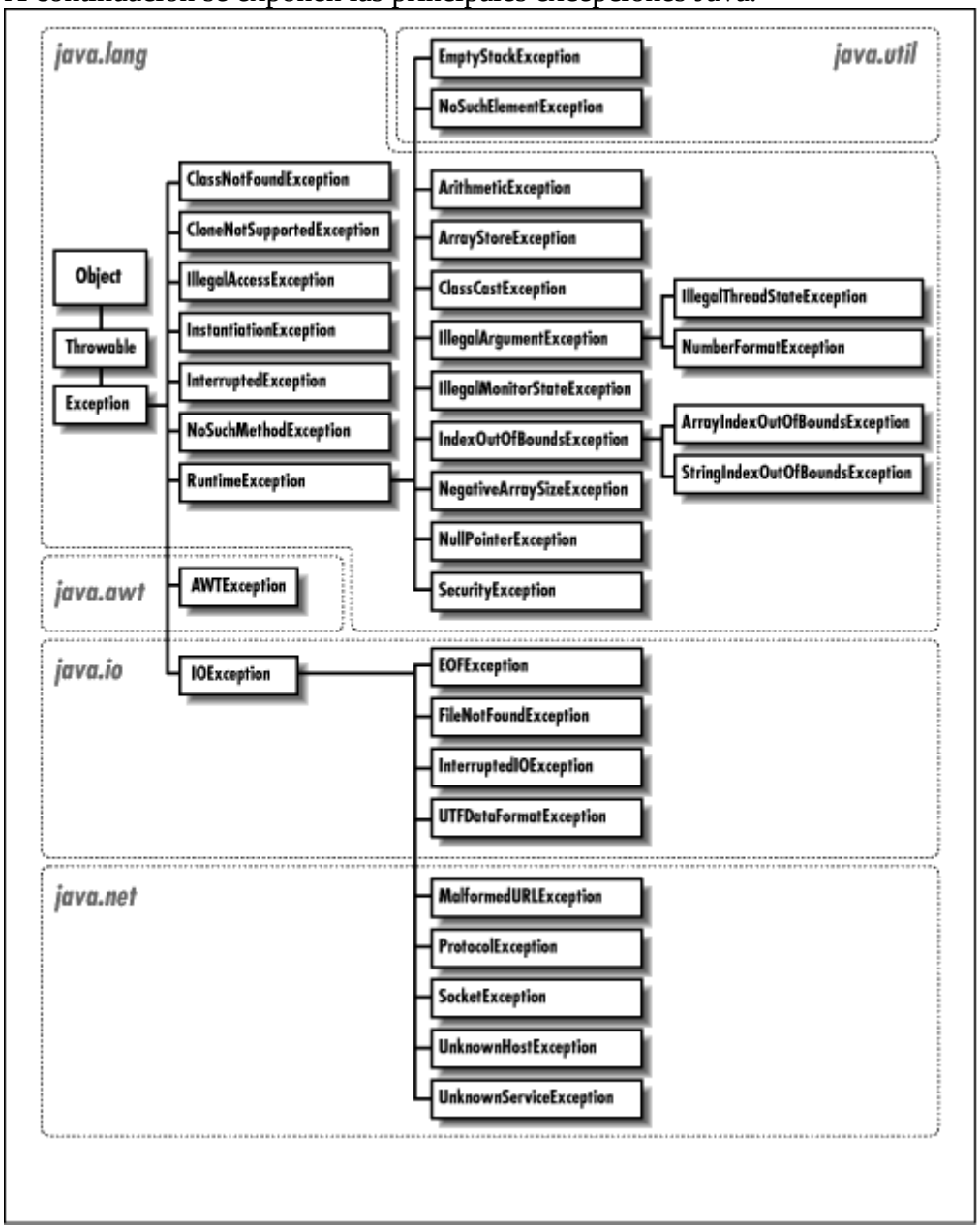
Una función JNI puede lanzar excepciones mediante la siguiente función.

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

El parámetro *class* inica la clase de la excepción que se va a lanzar. El parámetro *message* indica el mensaje descriptivo que contiene la excepción.
Si la excepción se lanza correctamente, retornará el valor 0. Si no se puede lanzar correctamente retornará un valor diferente de 0. Si ya hubiera una excepción lanzada, las siguientes darán error.
Si ya tenemos una referencia a la excepción, podemos lanzarla directamente mediante el uso de esta otra función.

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

A continuación se exponen las principales excepciones Java.



Ejemplo

Método nativo que recibe por parámetro un vector de enteros y una posición. Si la posición está dentro del vector, retornará el valor de esa posición. Si por el contrario es una posición errónea, lanzará una excepción.

```
package lanzaexcepciones;
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    public static native int cogeValor(int[]vector, int pos);
    public static void main(String[] args) {
        // TODO code application logic here
        int []vectorEnteros = {1, 1, 2, 3, 5, 8, 13, 21, 34, 55};
        int valor;
        try{
            valor = Main.cogeValor(vectorEnteros, 100);
        }catch(Exception ex)
        {
            System.out.println("Se ha producido una excepcion:
"+ex.getMessage());
            return;
        }
        System.out.println("El valor es: "+valor);
    }
    static{
        System.loadLibrary("LanzaExcepciones");
    }

}
```

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

#ifndef _Included_lanzaexcepciones_Main
#define _Included_lanzaexcepciones_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     lanzaexcepciones_Main
 * Method:    cogeValor
 * Signature: ([II)I
 */
JNIEXPORT jint JNICALL Java_lanzaexcepciones_Main_cogeValor
  (JNIEnv *, jclass, jintArray, jint);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "lanza.h"

JNIEXPORT jint JNICALL Java_lanzaexcepciones_Main_cogeValor
  (JNIEnv *env , jclass cls, jintArray vector, jint pos)
```

```
  {

      jsize vectorLen = (*env)->GetArrayLength(env, vector);
      if (pos>=vectorLen)
      {
            //Estoy fuera de rango, tengo que lanzar excepcion
            jclass classExcepcion = (*env)->FindClass(env,
"java/lang/ArrayIndexOutOfBoundsException");
            jint resul = (*env)->ThrowNew(env, classExcepcion,
"Excepcion en JNI. Posicion fuera de rango");
            if (resul==0)
                  printf("Excepcion lanzada\n");
      }
      jint* vectorC = (*env)->GetIntArrayElements(env, vector, NULL);
      jint retorno = vectorC[pos];
      (*env)->ReleaseIntArrayElements(env, vector, vectorC,
JNI_ABORT);
      return retorno;
  }
```

Programación multihilo en JNI

Restricciones JNI

Cuando llevemos a cabo la implementación de métodos nativos que pueden ser invocados por diferentes hilos, debemos tener encuenta las siguientes consideraciones: El puntero al entorno JNI es único para cada hilo. Nunca deberemos pasárselo a otro hilo ni cachearlo. El motivo es que este entorno JNI contiene datos internos al propio thread para asegurar compatibilidad con sistemas que carezcan de sopote para threads. Las referencias locales son sólo válidas para un hilo. Si queremos compartirlas, deberemos primero convertirlas en referencias globales.

Monitores

Los monitores son el principal mecanismo de sincronización utilizado en Java. Para poder aproderarnos de un monitor asociado a unu objeto en JNI tenemos las siguientes funciones.

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

La primera de ellas nos permite bloquear el monitor y la segunda liberarlo. Si la operación se ha llevado a cabo correctamente, se retornará un 0 (*JNI_OK*), y un valor diferente en caso contrario.
Pueden a su vez ser lanzadas dos excepciones.
OutOfMemoryError: Si ha habido un error de memoria.
IllegalStateException: Si hemos sido capaces de liberar un monitor que no teniamos bloqueado.

Ejemplo

Vamos a implementar el ejemplo del Master-Worker. En este caso la cola de los trabajos estará sincronizada mediante métodos nativos JNI.
Clase Master

```
package threadsjavajni;
import java.util.Vector;
public class Master implements Runnable{
    private Works _works;
    private int _counter;
    public Master(Works works) {
        _works = works;
        _counter = 0;
    }
   //---------------------------------------//
    private void insertaTrabajo(){
        _works.put(_counter);
        System.out.println("Master: Trabajo insertado "+_counter);
        _counter++;
    }
    //---------------------------------------//
    public void run(){
        while(_counter<5)
        {
            insertaTrabajo();
            try{
                //Thread.sleep(500);
```

```
            }catch(Exception ex){
            }
        }
        System.out.println("*****MASTER se retira");
    }
    //---------------------------------------//
}
```

Clase Worker

```
package threadsjavajni;
import java.util.Vector;
public class Worker implements Runnable {

    private Works _works;
    private boolean _fin;
    public Worker(Works work) {
        _works = work;
        _fin = false;

    }
    //---------------------------------------//
    private void cogeTrabajo(){
        try{
            int valor = _works.get();
            if (valor == -1)
                _fin = true;
            System.out.println("Trabajador: Cojo el trabajo "+valor);
        }catch(Exception ex)
        {
            _fin = true;
        }
    }
    //---------------------------------------//
    public void run(){
        while(!_fin)
        {
            cogeTrabajo();
            try{
                //Thread.sleep(1000);
            }catch(Exception ex){
            }
        }
        System.out.println("****WORKER se retira");
    }
    //---------------------------------------//
}
```

Clase Works

```
package threadsjavajni;
import java.util.Vector;
import java.io.*;
    public  class  Works{
      int _valor;
        //---------------------------------------//
        public Works(){
        }
        //---------------------------------------//
      public native int get();
```

```
        //---------------------------------------//
      public native void put(int valor);
        //---------------------------------------//
        static{
            System.loadLibrary("Works");
        }
        //---------------------------------------//
}
```

Clase principal Threads

```
package threadsjavajni;
import java.util.Vector;
public class Threads {
    public Threads() {
    }
    //---------------------------------------//
    public static void main(String []args){
        Works trabajos = new Works();
        Thread masterT = new Thread(new Master(trabajos));
        Thread workerT = new Thread(new Worker(trabajos));
        masterT.start();
        workerT.start();
    }
    //---------------------------------------//
}
```

Métodos nativos

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

#ifndef _Included_threadsjavajni_Works
#define _Included_threadsjavajni_Works
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     threadsjavajni_Works
 * Method:    get
 * Signature: ()I
 */
JNIEXPORT jint JNICALL Java_threadsjavajni_Works_get
  (JNIEnv *, jobject);

/*
 * Class:     threadsjavajni_Works
 * Method:    put
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_threadsjavajni_Works_put
  (JNIEnv *, jobject, jint);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "works.h"
JNIEXPORT jint JNICALL Java_threadsjavajni_Works_get
  (JNIEnv *env, jobject obj)
  {
      jint valor = -1;
      if ((*env)->MonitorEnter(env, obj)!=JNI_OK)
            return -1;
      //Estamos en seccion critica
      jclass clase = (*env)->GetObjectClass(env, obj);
      if(clase==NULL)
      {
      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return -1;
      }
      jfieldID campoID = (*env)->GetFieldID(env, clase, "_valor",
"I");
      if(campoID==NULL)
      {
      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return -1;
      }
      valor = (*env)->GetIntField(env, obj, campoID);
      (*env)->SetIntField(env, obj, campoID, (jint) -1);
      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return -1;
      return valor;
  }

/*
 * Class:     threadsjavajni_Works
 * Method:    put
 * Signature: (I)V
 */
JNIEXPORT void JNICALL Java_threadsjavajni_Works_put
  (JNIEnv *env, jobject obj, jint job)
  {
      if ((*env)->MonitorEnter(env, obj)!=JNI_OK)
            return;
      //Estamos en seccion critica
      //Estamos en seccion critica
      jclass clase = (*env)->GetObjectClass(env, obj);
      if(clase==NULL)
      {
      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return;
      }
      jfieldID campoID = (*env)->GetFieldID(env, clase, "_valor",
"I");
      if(campoID==NULL)
      {
      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return;
      }
      (*env)->SetIntField(env, obj, campoID, job);

      if ((*env)->MonitorExit(env, obj)!=JNI_OK)
            return;
  }
```

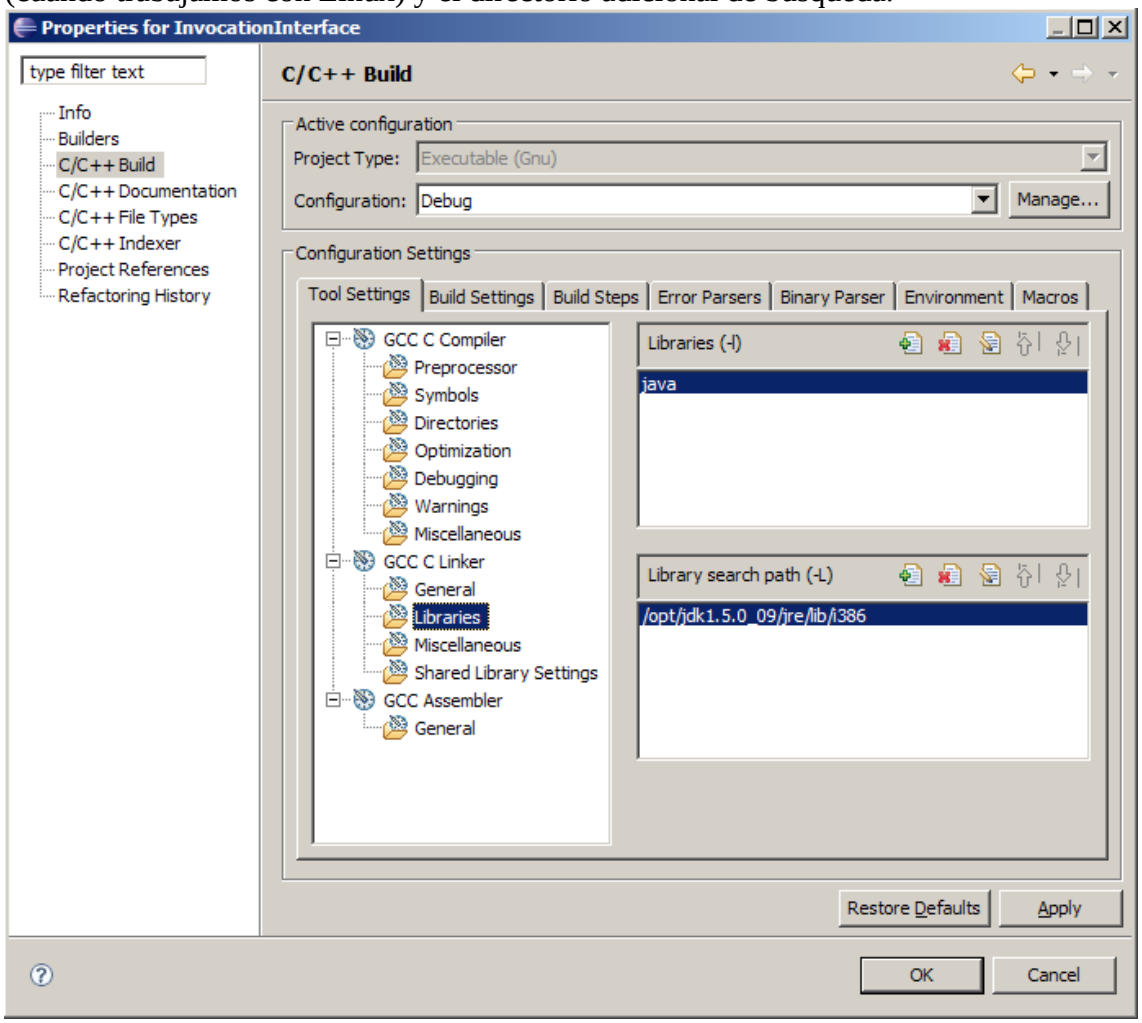
El Invocation Interface

En este apartado veremos cómo incrustar una máquina virtual Java en una aplicación nativa. Esta incrustación puede ser útil para los siguientes tipos de aplicaciones:
Aplicación nativa que ejecuta una aplicación Java.
Browser que ejecuta applets.
Servidor que permite 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”.

Consideraciones

Ahora que vamos a realizar aplicacions nativas de tipo ejecutable, necesitaremos indicarle al *linker* de *Eclipse* dónde encontrar las librerías dinámicas de la máquina virtual. Esto se lo indicaremos, dentro de las opciones del proyecto, en el apartado *“GCC C linker->Libraries”*. Una vez dentro debemos indicarle que use la librería *java* (cuando trabajamos con Linux) y el directorio adicional de búsqueda.



Creación de una máquina virtual

```
jint JNI CreateJavaVM(JavaVM** pvm, void** penv, void* vmargs)
```

Esta función nos permitirá crear una máquina virtual. Su resultado será 0 si todo está correcto.
El primer parámetro nos depositará el apuntador a la máquina virtua. El segundo nos depositará un apuntador al entorno JNI. El último parámetro será el que contenga las opciones de creación de la máquina virtual, y cuyo formato varía dependiendo si nos encontramos en JDSK 1.1 ó 1.2.
Para el JDSK 1.1 el formato es el siguiente.

```
typedef struct JDKllInitArgs {
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, valist args);
void (JNICALL *exit) (jint code);
void (JNICALL *abort) (void);
jint enableClassGC;
jint enableVerboseGC;
jint disableAsyncGC;
jint verbose;
jboolean debugging;
jint debugPort;
}
JDKllInitArgs;
```

En la siguiente tabla se describe el significado de cada campo.

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	Que bytescodes debe chequear la máquina virtual: O- 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. Util 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

También podemos obtener los valores por defecto.

```
void JNI GetDefaultJavaVMlnitArgs (void* vmargs)
```

Teniendo en cuenta que en el campo *version* siempre debe ir el valor *JNI_VERSION_1_1*.
Cuando trabajamos con la versión JSDK 1.2, la estructura de los argumentos es más sencilla, aunque no tenemos ninguna función para obtener los valores por defecto.

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

En este caso *version* deberá contener el valor *JNI_VERSION_1_2*. Donde *noptions* indica el número de opciones que tiene la estructura e *ignorUnrecognized* indica si ignorar cuando se encuentre un argumento no estándar y no reconocido (*JNI_TRUE*) o dar un error cuando esto suceda (*JNI_FALSE*).
El formato de las opciones se muestra a continuación.

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

El formato de las opciones suele ser *"-Dpropiedad=valor"*.
Además de la función de creación tenemos otras funciones para operar con la máquina virtual.

```
jint JNI GetCreatedJavaVMs(JavaVM** vmBuf, jsize bufLen, jsize* nvMs)
```

Obtiene las máquinas virtuales asociadas al proceso.

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

Descarga la máquina virtual. Cualquier hilo del proceso puede invocarla, pero esperará hasta que todos los hilos hayan finalizado su ejecución.

Cuando se crea una máquina virtual, sólo el hilo que la crea tiene acceso a ella. Un proceso puede tener asociados varios hilos, por lo que si queremos que tengan acceso a la máquina virtual, deberemos enlazarlos.

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

El parámetro de argumentos *args* puede ser *NULL* para compatibilidad con el JSDK 1.1, pero también puede tener la siguiente estructura.

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

De la misma forma que enlazamos, podemos desenlazar hilos de la máquina virtual.

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

Esta función desenlaza el hilo actual de la máquina virtual.
Por último, podemos obtener el entorno de un hilo.

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

Esto nos permitirá saber si dicho hilo está asociado o no a la máquina virtual.
Por último, tenemos dos funciones que si son exportadas e implementadas, se ejecutarán en la carga y la descarga de la máquina. Esto nos permitirá ejecutar otras rutinas en el mismo momento en que la máquina virtual es cargada o es descargada.

```
JNIEXPORT jint JNICALL JNIOnLoad(JavaVM* jvm, void* reserved)
JNIEXPORT void JNICALL JNIOnUnload(JavaVM* jvm, void* reserved)
```

Ejemplo 1

Programa que crea una máquina virtual y ejecuta una aplicación
Aplicación Java a ejecutar.

```
public class Main {

    /** Creates a new instance of Main */
    public Main() {
    }

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        System.out.println("Hola!!! "+args[0]);
    }

}
```

Código nativo.

```
#include <jni.h>
   #define USER_CLASSPATH "."
   #include <dlfcn.h>

   int main(int argc, char* argv[]) {

       JNIEnv *env;
       JavaVM *jvm;
       jint res;
       if(argc!=2)
       {
                   printf("Debe pasar la clase a ejecutar por
parametro\n");
                   printf("Ejemplo: invocationinterface clase\n");
                   return(-1);
       }

       JavaVMInitArgs vm_args;
            JavaVMOption options[1];
            options[0].optionString ="-
Djava.class.path=/opt/jdk1.5.0_09/lib:.:./Debug";
       options[0].extraInfo = NULL;
       vm_args.version = JNI_VERSION_1_2;
       vm_args.options = options;
       vm_args.nOptions = 1;
       vm_args.ignoreUnrecognized = JNI_FALSE;
       // Create the Java VM
       res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
       if (res < 0) {
            fprintf(stderr, "Can't create Java VM\n");
            return(1);
       }
       printf("Resultado de crear la vm %d\n", res);
       jclass cls = (*env)->FindClass(env, argv[1]);
       if (cls == NULL) {
                   printf("No se encuentra la clase %s\n", argv[1]);
            goto destroy;
       }
```

```
            jmethodID mid = (*env)->GetStaticMethodID(env, cls,
"main","([Ljava/lang/String;)V");
            if (mid == NULL) {
                        printf("La clase no contiene metodo main\n");
                        goto destroy;
            }

            jstring jstr = (*env)->NewStringUTF(env, " desde JNI!");
            if (jstr == NULL) {
                  goto destroy;
            }
            jclass stringClass = (*env)->FindClass(env,
"java/lang/String");
            jobjectArray args = (*env)->NewObjectArray(env, 1,
stringClass, jstr);
            if (args == NULL) {
                  goto destroy;
            }
            (*env)->CallStaticVoidMethod(env, cls, mid, args);

destroy:
                  if ((*env)->ExceptionOccurred(env)) {
                        //(*env)->ExceptionDescribe(env);
                  }
                  (*jvm)->DestroyJavaVM(jvm);
            return(0);
}
```

Ejemplo 2

Programa que ejecuta una clase. Esta clase a su vez llamará a un método nativo que invocará un método Java.

Aplicación Java.

```
package metodosinstancia;
public class Main {
    private int resultado;
    public Main() {
    }

    public int getResultado(){
        return resultado;
    }
    public void suma(int a, int b){
        resultado = a + b;
    }
    public native void sumaC();
    public static void main(String[] args) {
        Main metodos = new Main();
        System.out.println("Antes de JNI resultado es: "
+metodos.getResultado());
        metodos.sumaC();
```

```
        System.out.println("Despues de JNI resultado es: "
+metodos.getResultado());
    }
    static {
        System.loadLibrary("MetodosInstancia");
    }
}
```

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

#ifndef _Included_metodosinstancia_Main
#define _Included_metodosinstancia_Main
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     metodosinstancia_Main
 * Method:    sumaC
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_metodosinstancia_Main_sumaC
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
#include "metodos.h"

JNIEXPORT void JNICALL Java_metodosinstancia_Main_sumaC
  (JNIEnv *env,  jobject obj)
  {
      jclass clase = (*env)->FindClass(env, "metodosinstancia/Main");
      if (clase==NULL)
            return;
      jmethodID metodoID = (*env)->GetMethodID(env, clase, "suma",
"(II)V");
      if (metodoID==NULL)
            return;
      (*env)->CallVoidMethod(env, obj, metodoID, 10, 15);
  }
```

Aplicación nativa.

```
   #include <jni.h>
   #define USER_CLASSPATH "."
   #include <dlfcn.h>

   int main(int argc, char* argv[]) {

        JNIEnv *env;
        JavaVM *jvm;
        jint res;
        if(argc!=2)
        {
```

```
                    printf("Debe pasar la clase a ejecutar por
parametro\n");
                    printf("Ejemplo: invocationinterface clase\n");
                    return(-1);
        }

        JavaVMInitArgs vm_args;
             JavaVMOption options[2];
          options[0].optionString ="-
Djava.class.path=/opt/jdk1.5.0_09/lib:.:/home/javi/cursojni/netbeans/M
etodosInstancia/build/classes";
        options[0].extraInfo = NULL;
        options[1].optionString="-
Djava.library.path=/home/javi/cursojni/eclipse/MetodosInstancia/Debug"
;
        options[1].extraInfo = NULL;
        vm_args.version = JNI_VERSION_1_2;
        vm_args.options = options;
        vm_args.nOptions = 2;
        vm_args.ignoreUnrecognized = JNI_FALSE;
        // Create the Java VM
        res = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
        if (res < 0) {
             fprintf(stderr, "Can't create Java VM\n");
             return(1);
        }
        printf("Resultado de crear la vm %d\n", res);
        jclass cls = (*env)->FindClass(env, argv[1]);
        if (cls == NULL) {
                    printf("No se encuentra la clase %s\n", argv[1]);
             goto destroy;
        }

             jmethodID mid = (*env)->GetStaticMethodID(env, cls,
"main","([Ljava/lang/String;)V");
             if (mid == NULL) {
                        printf("La clase no contiene metodo main\n");
                        goto destroy;
             }

             jstring jstr = (*env)->NewStringUTF(env, " desde JNI!");
             if (jstr == NULL) {
                    goto destroy;
             }
             jclass stringClass = (*env)->FindClass(env,
"java/lang/String");
             jobjectArray args = (*env)->NewObjectArray(env, 1,
stringClass, jstr);
             if (args == NULL) {
                    goto destroy;
             }
             (*env)->CallStaticVoidMethod(env, cls, mid, args);

destroy:
                    if ((*env)->ExceptionOccurred(env)) {
                          (*env)->ExceptionDescribe(env);
                    }
                    (*jvm)->DestroyJavaVM(jvm);
             return(0);
}
```

Bibliografía

"***The Java™ Native Interface. Programmer's Guide and Specification***"
Sheng Liang
ADDISON-WESLEY

"***Thinking in Java, 2nd Edition***"
Bruce Eckel
Prentice-Hall

Webliografía

"***JavaTM 2 Platform Standard Edition 5.0 API Specification***"
http://java.sun.com/j2se/1.5.0/docs/api/

"***JNI Specification***"
http://java.sun.com/j2se/1.3/docs/guide/jni/spec/jniTOC.doc.html

"***Java Native Interface***"
http://java.sun.com/j2se/1.4.2/docs/guide/jni/index.html

"***Java Native Interface 1.5 Specification***"
http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html