INVOCAR PROGRAMAS ASM DESDE JAVA PASO A PASO

Contenido

1. Introducción

En el sitio de java.net existe un pequeño tutorial titulado Invoking Assembly Language Programs from Java[1] pero está en inglés y no es muy sencillo para algunos nuevos programadores Java o ensamblador. En java.net explican cómo funciona la invocación de un programa ASM desde Java y qué hacer para lograrlo, pero no explican cómo hacer cada paso. Por lo tanto me he dado a la tarea de facilitar la comprensión de estas técnicas para los hispanohablantes que dan sus primeros pasos con JNI; ya que conocer un poco del funcionamiento de JNI y la invocación de aplicaciones en ensamblador es fundamental cuando se tienen aplicaciones que necesiten operaciones que consuman mucho proceso del servidor y se necesite implementar un pequeño programa a bajo nivel para ejecutar tales operaciones, y de esta manera disminuir el tiempo de espera que a veces suele ser crítico. También con JNI tenemos la capacidad de invocar funciones y procedimientos escritos en otros lenguajes, como C/C++ o en nuestro caso ASM, por ejemplo, cuando se tienen sistemas legados y se necesite utilizar alguna función del código legado será necesario desarrollar aplicaciones que puedan ejecutar estas funciones. Para más información de JNI y sus capacidades se puede consultar el manual para programadores disponible en el sitio de Sun[2].

Cabe señalar que este no es un tutorial introductorio a JNI o al API, por lo tanto se limitará a mencionar solo algunos conceptos necesarios. De forma rápida, comenzaremos con determinar el software que usaremos y de dónde obtenerlo, pero no nos detendremos a aprender el funcionamiento de ellos. Posteriormente agregaremos algunas variables de entorno y comenzaremos con la programación. Para esto, se presentan los programas y cómo compilarlos, sin embargo estará disponible el código fuente usado en el tutorial. Al finalizar el tutorial el lector deberá ser capaz de cargar una librería DLL en cualquier aplicación Java y usar los procedimientos de cualquier librería nativa.

2. ¿Cuándo usar JNI?

En algunas ocasiones, a algunos desarrolladores les tocará encontrarse en situaciones en las que una aplicación hecha completamente en Java no cubrirá todos los requerimientos para tal. Algunos ejemplos de estas situaciones pueden ser:

- Cuando a pesar de querer escribir toda la aplicación en Java para permitir la compatibilidad entre plataformas, exiten características de la plataforma que son necesarias para la aplicación y no son soportadas por la librería estándar de Java. Esto es referido como dependencia de la plataforma.

- Cuando ya se tiene una librería escrita en algún otro lenguaje y se desea hacerla accesible a nuestro código Java, por ejemplo, al tener que trabajar con código legacy. en estos casos las librerías se cargan dentro del mismo proceso de la apliación por medio de JNI, aunque existen otros mecanismos más eficientes que funcionan en procesos separados.
- Si se quiere implementar porciones de código en un lenguaje de bajo nivel como ensamblador para disminuir el tiempo de procesamiento. Por ejemplo, en aplicaciones que necesiten renderizar gráficos 3D que requieren más tiempo de procesamiento, habrá que escribir una librería para gráficos en lenguaje ensamblador para tener un mejor rendimiento[3].
- En casos en los que quiera cargar una librería nativa en un proceso existente para evitar el costo de iniciar un nuevo proceso y cargar la librería en el mismo[4].
- También será oportuno usar JNI cuando se quiera utilizar algunas funcionalidades de un programa Java desde un código nativo.

3. Requerimientos de software

Para desarrollar los ejemplos se estará trabajando sobre un sistema operativo Windows XP de 32 bits. El software adicional que se usará son los siguientes:

1. Editor. Servirá para editar el código Java y el código del programa ensamblador (ASM). Para esto puede ser de gran utilidad cualquier editor de texto o algunos más sofisticados como Notepad++[5] , o IDEs especializados. Yo uso Eclipse con un plugin para editar archivos ASM, pero también funcionan NetBeans y el VisualStudio.
2. MASM32[6]. Se necesita para compilar los programas ASM.
3. Java[7]. Se debe tener instalado alguna versión del JDK de Java; preferentemente versión 4 o posterior para evitar posibles incompatibilidades en nuestro código.

4. Configuración del ambiente de trabajo

Antes de comenzar a programar será oportuno asegurarse de tener algunas variables entorno[8] necesarias para poder ejecutar algunas instrucciones desde la ventana de comandos sin la necesidad de almacenar el código dentro de los directorios que contienen los comandos/programas que serán invocados.

Pasos generales para agregar variables de entorno.

- Presionar Win+Pausa para abrir las propiedades del sistema.
- En la ficha 'Opciones avanzadas' clic en el botón 'Variables de entorno' (*Imagen 1*).

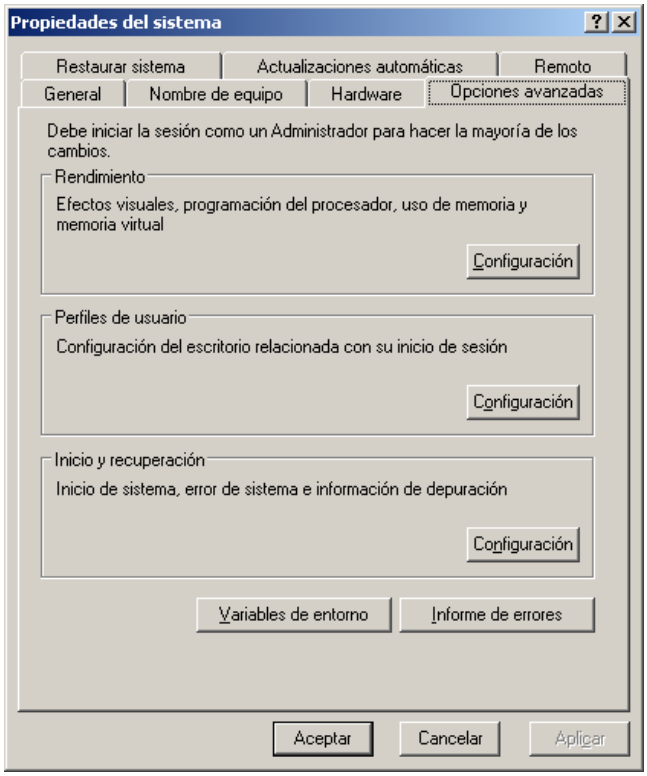

Imagen 1. Ficha Opciones avanzadas de Propiedades del sistema

- En la nueva ventana (*Imagen 2*) clic en el botón 'Nueva' de la sección 'Variables del sistema'.

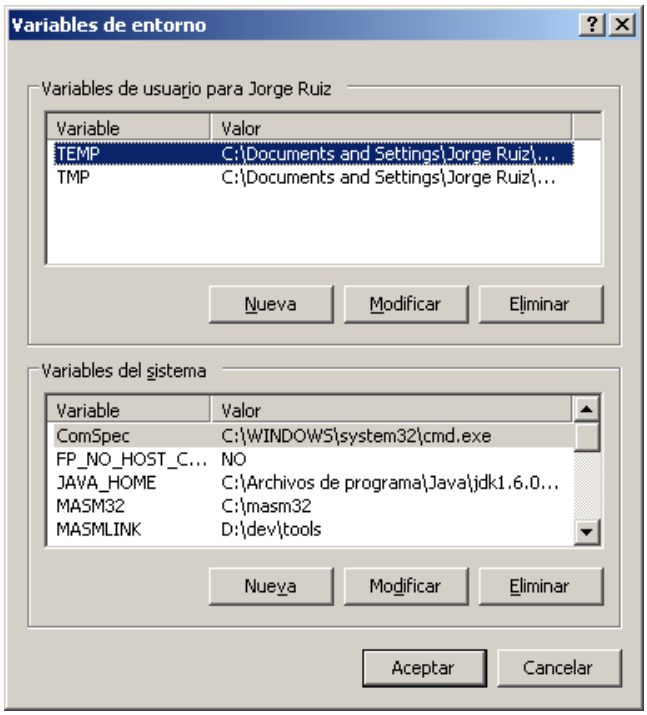

Imagen 2. Variables de entorno

- En la ventana emergente 'Nueva variable del sistema' (*Imagen 3*) poner el nombre de la variable y la ubicación de la carpeta de la variable.

Imagen 3. Ventana de agregar Nueva variable de sistema

Se procederá a agregar las siguientes variables de entorno:

1. Nombre de variable: JAVA_HOME
- Valor: La dirección donde se encuentra instalado Java (C:\Archivos de programa\Java\jdk1.5.0_04)
2. Nombre de variable: MASM32
- Valor: La dirección donde se encuentra instalado Masm32 (C:\masm32)
3. Edite la variable 'Path' agregándole lo siguiente al valor de la variable:
- ;%JAVA_HOME%\bin;%MASM32%\bin
4. Para asegurarse que se han agregado correctamente las variables de entorno, ejecutar las siguientes instrucciones:
- **`java -version`** para verificar la versión de Java que está siendo usada
- **`ml`** para asegurarse que reconoce el comando correspondiente al compilador de Masm32. Dado que el comando no tiene una opción para mostrar solo la versión del compilador, lo que se desplegará en la pantalla será un error que previamente mostrará la versión del compilador.
- **`link /`** con la diagonal, como si fuese a agregar parámetros, para evitar que se despliegue en la pantalla las opciones del linker. De igual manera, se verá en la pantalla un error que previamente mostrará la versión del enlazador.

En la *Imagen 4* se puede apreciar el resultado de las ejecuciones una tras otra. Si en vez de obtener los resultados esperados como se observan en la imagen se lanza un error con el mensaje *"<comando>" no se reconoce como un comando interno o externo, programa o archivo por lotes ejecutable*, entonces probablemente se ha agregado incorrectamente la variable de entorno y deberá ser necesario volver sobre los pasos para hallar el error.

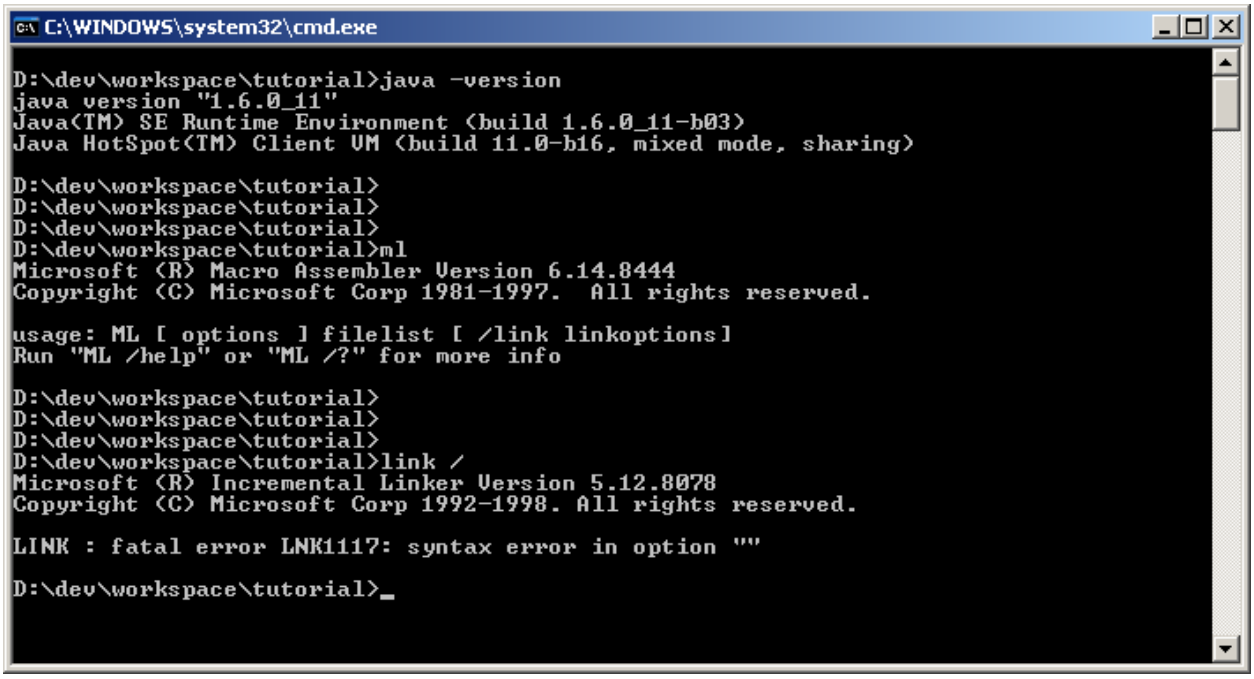

Imagen 4. Comprobación de las variables de entorno

Luego de haber agregado las variables de entorno se selecciona 'Aceptar' en las dos ventanas emergentes anteriores para guardar todos los cambios.

5. La programación

Ahora todo está listo y ya se puede comenzar a programar. Pero antes de hacerlo, primero deberemos poder hacer una recreación mental del diseño de nuestra aplicación y los pasos que deberemos seguir para lograrlo. De forma general y rápida, en la *Imagen 5* vemos un pequeño diseño de la estructura de la aplicación y la dinámica de las invocaciones.

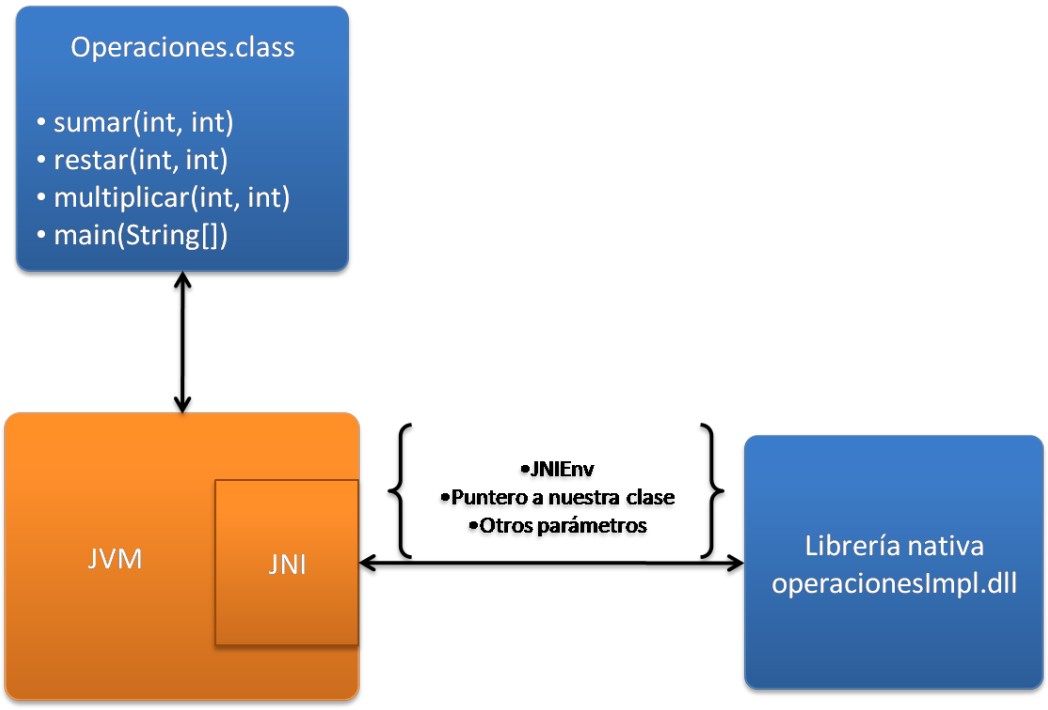

Imagen 5. Dinámica de la invocación de métodos nativos con JNI

En la *Imagen 6* apreciamos un diagrama de los pasos a llevar a cabo para crear y poner en marcha nuestro ejemplo. Para comprender en qué consiste la técnica que usaremos explicaremos rápidamente cada uno de los pasos que deberemos llevar a cabo.

1. Crear la clase que declara los métodos nativos. En este paso es donde hacemos nuestra aplicación Java o simplemente escribimos la clase donde hemos de declarar todos los métodos nativos que usaremos en nuestra aplicación. Puede ser, por ejemplo, una sola clase que se dedique exclusivamente para este efecto.
2. Usar `javac` para compilar el programa. Una vez que se tiene la clase que declara los métodos nativos, es necesario compilarla. En este punto deberemos detenernos un poco para entender que aunque no existan las librerías que contienen los métodos nativos, Java podrá compilar el código, pero de ninguna manera se podrá ejecutar la aplicación porque ocurrirá un error en tiempo de ejecución al intentar cargar la librería con tales funciones.
3. Usar `javah -jni` para generar la cabecera de C. Este paso es crucial, ya que con esta instrucción podremos generar un archivo `.h` que nos indicará el nombre y los parámetros con los que deberán ser nombradas y parametrizadas cada una de las funciones que vayamos a implementar en lenguaje ensamblador.
4. Escribir el programa en ensamblador que implemente los métodos nativos. Con la especificación de los nombres solo queda implementar nuestra funcionalidad.
5. Usar `masm32` para compilar el código nativo. Entre todas las herramientas que contiene el programa masm32, necesitaremos especialmente el que nos ayuda a compilar programas ASM con el que ensamblaremos un archivo objeto con el que más adelante obtendremos nuestra librería nativa.
6. Usar la herramienta `masm32` para generar la librería nativa. Finalmente, para tener todos los archivos necesarios para ejecutar nuestra aplicación, deberemos generar la librería nativa, en este caso un DLL, que cargará nuestro programa Java. Para llevar a cabo este paso también se usará una herramienta disponible en masm32.

7. Correr la aplicación usando el intérprete de Java. Ahora, llegando a este paso, ya podremos ejecutar nuestra aplicación como lo haríamos con cualquier aplicación Java, tomando en cuenta que deberemos tener la librería nativa que hemos generado en la ubicación que hemos definido en la clase que carga dicha librería.

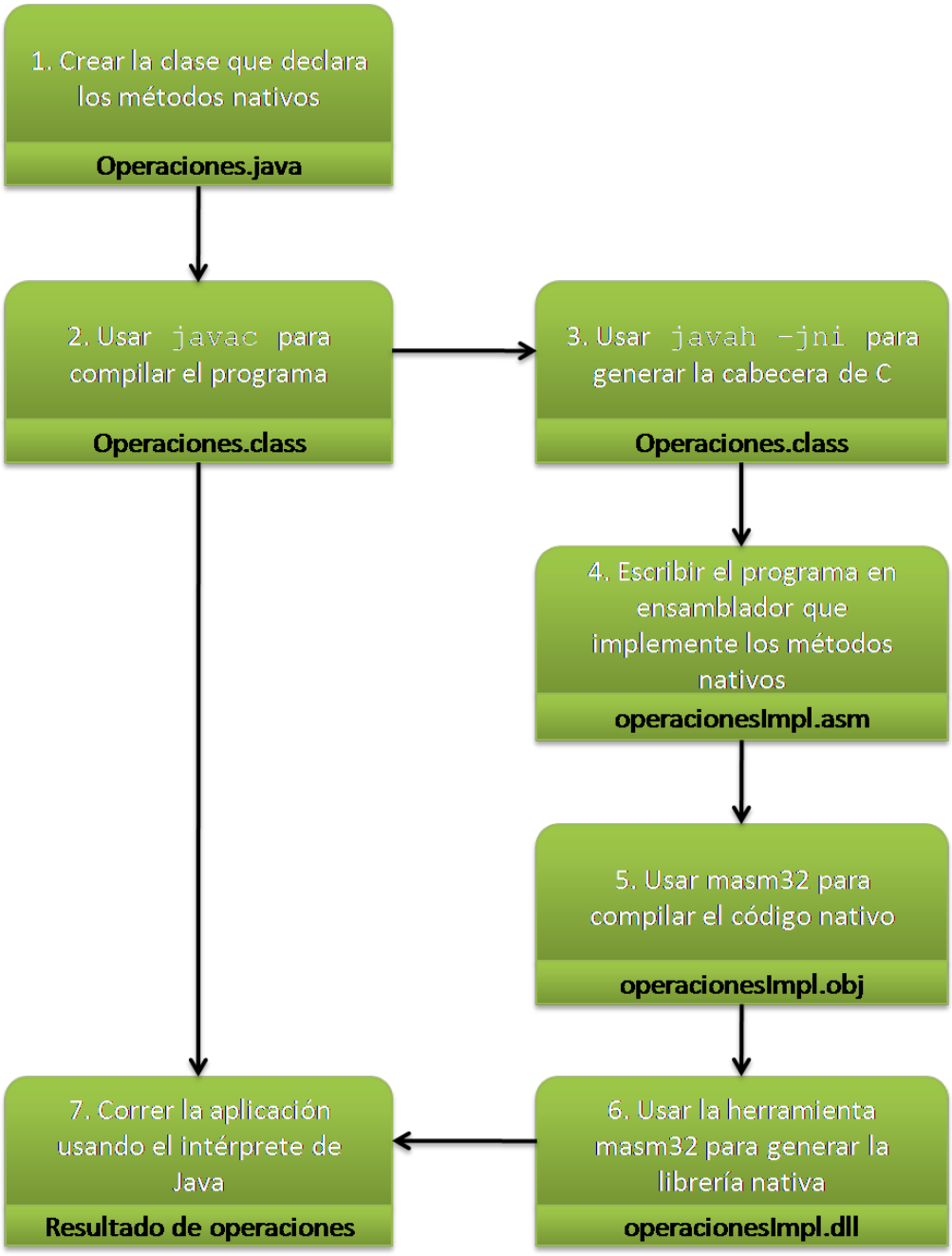

Imagen 6. Pasos para lograr que una aplicación Java use una librería nativa

Ahora procederemos a crear nuestro programa en Java llamado Operaciones.java y lo guardamos en el directorio de nuestra preferencia, donde también guardaremos los demás archivos que crearemos. Operaciones.java carga una librería nativa que contiene el código necesario para realizar la suma, resta y multiplicación de dos números dados (n1 y n2), y el

resultado devuelto es impreso en la pantalla desde nuestra aplicación Java.

Paso 1. Crear la clase que declara los métodos nativos: Operaciones.java

```
public class Operaciones {
    public native int sumar(int a, int b);
    public native int restar(int a, int b);
    public native int multiplicar(int a, int b);

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

    public static void main(String[] args) {
        int n1 = 2;
        int n2 = 3;
        Operaciones op = new Operaciones();
        int resultado = op.sumar(n1, n2);
        System.out.println("El resultado de la suma es: " +
resultado);

        resultado = op.restar(n1, n2);
        System.out.println("El resultado de la resta es: " +
resultado);

        resultado = op.multiplicar(n1, n2);
        System.out.println("El resultado de la multiplicacion es: "
+ resultado);
    }
}
```

- **`public native int sumar(int a, int b);`** En esta línea declaramos un método nativo que recibirá dos parámetros enteros y devolverá un entero que es el resultado de la suma. Como este, hay dos métodos más declarados, **restar** y **multiplicar**.
- **`System.loadLibrary("operacionesImpl");`** Se le indica al JVM cuál será el nombre de la librería en la cual se encuentra el método que deseamos invocar, en nuestro caso se llamará *operacionesImpl*.

Pasos 2 y 3.
Usar `javac` para compilar el programa.
Usar `javah -jni` para generar la cabecera de C

Ahora, para poder crear nuestro programa ASM necesitaremos saber cómo llamaremos a nuestro procedimiento dentro del programa, para esto JNI hace algo llamado decorado de nombres en el cual concatena con signos de guión bajo (_) con los siguientes datos: por default primero aparece la palabra "Java" seguido del nombre de la clase y finalmente el nombre del método y los parámetros del mismo. Pero esto es más comprensible cuando creamos la cabecera de C++, que nos dirá como deberá llamarse nuestro procedimiento. Para hacer esto realizamos los siguientes pasos:

1. Abrimos la ventana de comandos y entramos al directorio donde hemos guardado el archivo Operaciones.java.
2. Ejecutamos `javac Operaciones.java` (*Imagen 7*)
3. Esto habrá creado un archivo .class llamado Operaciones.class
4. Ejecutamos enseguida `javah Operaciones` (*Imagen 7*)
5. Esto habrá generado, usando el archivo Operaciones.class, el archivo Operaciones.h el cual contiene las cabeceras de los procedimientos que necesitaremos implementar en ASM

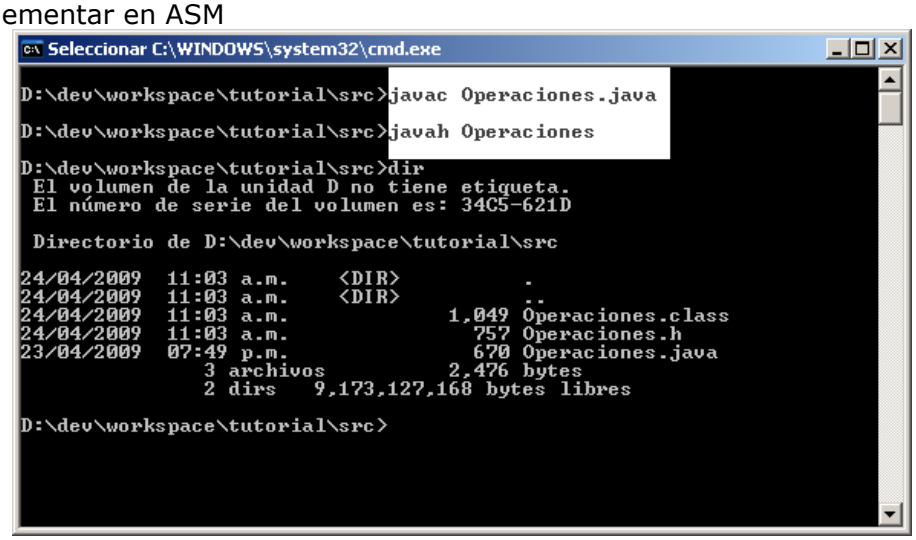

Imagen 7. Generación del .class y el .h a partir del archivo java

Hasta ahora tenemos los siguientes archivos en nuestro directorio de trabajo:

- *Operaciones.java*
- *Operaciones.class*
- *Operaciones.h*

Si han ocurrido errores al momento de compilar deberemos leer de qué se trata, posiblemente el error pueda ser causa de que las variables de entorno no estén correctamente definidas y por tanto el MSDOS no logre encontrar el comando, o también se deba a algún error en el código escrito. Y una vez que el programa sea compilado sin errores abra el archivo *Operaciones.h* con el editor de código y verá un programa en C++ como este:

Operaciones.h

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

#ifndef _Included_Operaciones
#define _Included_Operaciones
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Operaciones
 * Method:    sumar
```

```
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_Operaciones_sumar
  (JNIEnv *, jobject, jint, jint);

/*
 * Class:     Operaciones
 * Method:    restar
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_Operaciones_restar
  (JNIEnv *, jobject, jint, jint);

/*
 * Class:     Operaciones
 * Method:    multiplicar
 * Signature: (II)I
 */
JNIEXPORT jint JNICALL Java_Operaciones_multiplicar
  (JNIEnv *, jobject, jint, jint);

#ifdef __cplusplus
}
#endif
#endif
```

No es necesario conocer de programación en C++, solo basta con entender que lo que está en negrita es como deberemos llamar a nuestro procedimiento en ASM y los parámetros que deberá recibir. En este caso tendremos tres procedimientos llamados **Java_Operaciones_sumar**, **Java_Operaciones_restar** y **Java_Operaciones_multiplicar**, cada uno con cuatro parámetros. Los primeros dos parámetros se generan por default; **JNIEnv** es un puntero hacia las funciones que se pueden usar para interactuar con la JVM y objetos; **jobject** hace referencia a nuestro propio objeto; y las más importantes que debemos tener en cuenta son dos parámetros **jint** que nos indican que está recibiendo dos parámetros de tipo entero[9]. Además, estaremos devolviendo un parámetro de tipo entero (jint). Ahora que ya sabemos cómo se llamará o llamarán los procedimientos que deseemos definir, procederemos a crear el programa ASM y lo llamaremos *operacionesImpl.asm*.

Paso 4. Escribir el programa en ensamblador que implemente los métodos nativos: operacionesImpl.asm

```
.386
.model flat,stdcall

.code

Java_Operaciones_sumar proc JNIEnv:DWORD, jobject:DWORD, a:DWORD,
b:DWORD
    mov eax, a    ;ponemos el valor de a en el registro eax
    mov ebx, b    ;ponemos el valor de b en el registro ebx
```

```
    add eax,ebx    ;sumamos el contenido del registro eax con el de
ebx y
                ;el resultado se guarda en eax
    ret    ;por default se retorna el valor del registro eax
Java_Operaciones_sumar endp

Java_Operaciones_restar proc JNIEnv:DWORD, jobject:DWORD, a:DWORD,
b:DWORD
    mov eax, a
    mov ebx, b
    sub eax, ebx    ;restamos el contenido del registro ebx al de
abx
    ret
Java_Operaciones_restar endp

Java_Operaciones_multiplicar proc JNIEnv:DWORD, jobject:DWORD,
a:DWORD, b:DWORD
    mov eax, a
    mov ebx, b
    mul ebx    ;multiplicamos el contenido del registro ebx con el
registro por default eax
    ret
Java_Operaciones_multiplicar endp

END
```

- **.386** define el conjunto de instrucciones que estaremos usando, aquí 80386
- **.model flat** indica que estaremos usando un tipo de memoria plana de 32bits que es el que usa el 386
- **stdcall** define el orden en que se van a pasar los parámetros (izquierda a derecha o derecha a izquierda)
- **Java_Operaciones_sumar proc JNIEnv:DWORD, jobject:DWORD, a:DWORD, b:DWORD**. Aquí podemos ver la declaración de nuestro procedimiento donde se define que se llamará Java_Operaciones_sumar, tal como lo vimos antes, y tendrá cuatro parámetros. Aquí podemos nombrar a nuestros parámetros como nos plazca, pero debemos tener en cuenta el orden en el que aparecen y el tipo de dato, en nuestro caso los parámetros que nos importan se llaman **a** y **b**.
- El resto es el código ASM que realizará la tarea que deseamos, en este caso sumar a y b
- **ret** al terminar de ejecutarse nuestro código, se hará un return que devolverá el contenido del registro por default **eax**, por eso no se indica explícitamente. El valor del registro eax será el que capturará el programa Java y lo mostrará en la salida de pantalla

Pasos 5. Usar `masm32` para compilar el código nativo

El siguiente paso será compilar este código con MASM32 y generar los archivos .obj y .dll, y con ellos otros más que no usaremos por el momento.
Para generar el archivo *operacionesImpl.obj*:

1. Acceder al directorio donde tenemos el arhivo *operacionesImpl.asm*
2. Ejecutar la siguiente línea para crear el archivo *operacionesImpl.obj* (*Imagen 8*)

```
ml /c /coff /Cp operacionesImpl.asm
```

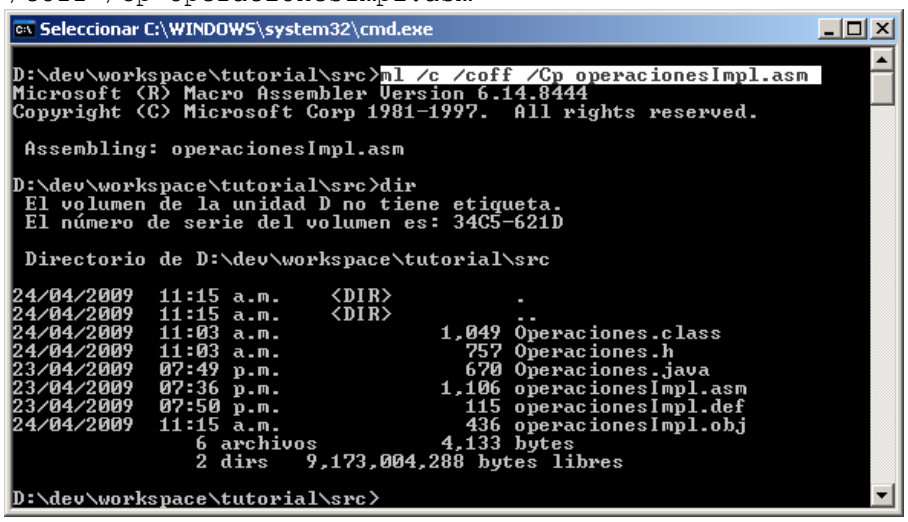

Imagen 8. Compilación del programa ensamblador

La explicación:

- `ml` es el programa de MASM32 para crear el objeto
- `/c` le indicamos que solo habrá de ensamblar el .obj
- `/coff` para indicar que el objeto ensamblado tendrá el formato COFF (Common Object File Format)
- `/Cp` indicará a MASM32 que será sensible a mayúsculas y minúsculas de los identificadores que se usen

Paso 6. Usar la herramienta `masm32` para generar la librería nativa

1. Ahora deberemos crear un archivo de definición para generar nuestra librería dinámica, para esto crearemos un archivo de texto con extensión .DEF llamado operacionesImpl.def, aunque el nombre puede ser cualquier otro, pero usaremos este nombre para nuestro ejemplo.

operacionesImpl.def

```
LIBRARY operacionesImpl
EXPORTS
    Java_Operaciones_sumar
    Java_Operaciones_restar
    Java_Operaciones_multiplicar
```

- **`LIBRARY`** declara el nombre que tendrá el DLL. En nuestro ejemplo lo llamaremos *sumar* porque así lo tenemos en el código del programa *Operaciones.java*
- **`EXPORTS`** indica el nombre de las funciones o procedimientos que serán exportados en el DLL, en el tutorial solo tenemos el procedimiento Java_Operaciones_sumar

2. Una vez habiendo guardado suma.def con los demás archivos seguiremos con la ejecución de una instrucción más donde se podrá observar el uso de los archivos *operacionesImpl.def* y *operacionesImpl.obj*. (*Imagen 9*)

```
link /DLL /NOENTRY /subsystem:windows /DEF:operacionesImpl.def
operacionesImpl.obj
```

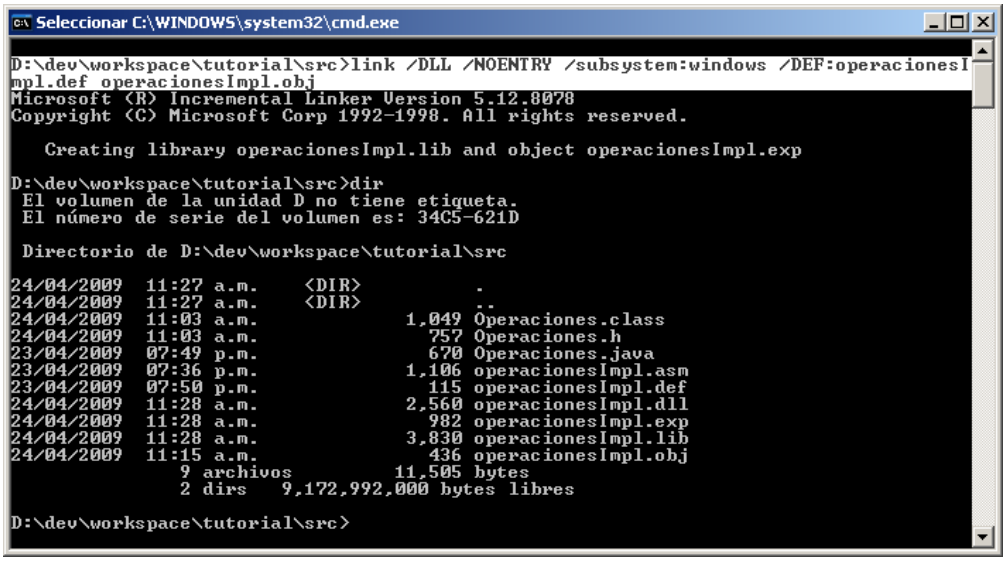

Imagen 9. Generación del linker

La explicación:

- `link` el programa (linker) que usaremos para generar el .dll
- `/DLL` indica que lo que queremos generar es un DLL
- `/NOENTRY` para poder evitar algunos posibles errores no capturados
- `/SUBSYSTEM:windows` indicamos que el ejecutable es para windows
- `/DEF:archivo.def` y el archivo de definición que se usará para la exportación
- Finalmente indicamos el nombre de nuestro archivo ensamblado, *operacionesImpl.obj*

3. Ahora deberemos tener en nuestro directorio los archivos mostrados en la siguiente lista:

- *Operaciones.class*
- *Operaciones.h*
- *Operaciones.java*
- *operacionesImpl.asm*
- *operacionesImpl.def*
- *operacionesImpl.dll*
- *operacionesImpl.exp*
- *operacionesImpl.lib*
- *operacionesImpl.obj*

6. La ejecución

Finalmente tenemos todos los archivos necesarios, pero para ejecutar nuestra aplicación solo serán necesarios los archivos *Operaciones.class* y *operacionesImpl.dll* (*Imagen 10*). Procedemos finalmente a ejecutar la siguiente instrucción:

```
java Operaciones
```

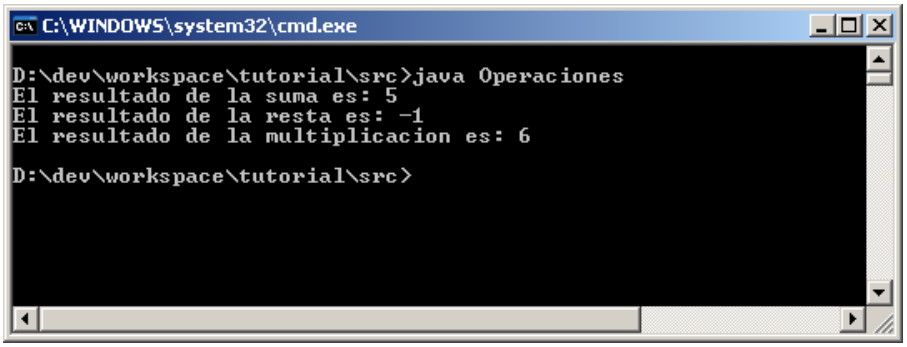

Imagen 10. Funcionamiento de nuestro programa

Si se han seguido correctamente todos los pasos, habremos podido ejecutar una aplicación Java en la cual invocamos un procedimiento escrito en lenguaje ensamblador.

7. Conclusión

Hasta ahora hemos aprendido lo fácil que es invocar funciones nativas programadas en ensamblador desde nuestras aplicaciones Java. Sin embargo, aunque nuestro ejemplo fue muy sencillo ya que solamente implementamos e invocamos funciones con operaciones matemáticas básicas, este es el procedimiento general para hacer cualquier tipo de invocaciones, pero es importante tener en cuenta que los parámetros de los métodos pueden cambiar, así que también cambiarán los parámetros que reciban nuestras funciones. Lo que queda por delante, a quienes les interese saber más acerca de los usos de JNI, es aprender cómo usar librerías del sistema operativo para que nuestras funciones realicen tareas más complejas y completas. Por ejemplo, para poder mostrar ventanas y otros tipos de gráficos desde nuestras funciones en ensamblador; una práctica común para esto es implementar una calculadora como el de Windows que también use el coprocesador de punto flotante[10]. También puede intentar mejorar el rendimiento de alguna aplicación que realice operaciones matemáticas complejas, invocando operaciones implementadas en ensamblador. O tal vez prefiera probar usar funciones de un sistema legacy, si tiene acceso a uno. Por otro lado, si usa otros sistemas operativos como Lunix, Mac, etc. los pasos que se han descrito son los mismos para lograr resultados similares, solo que en cada caso se deberá preveer que las herramientas y el tipo de librería nativa serán disitintas. Lo que me queda decirle es que hay muchas ventajas que puede obtener de aprender estas técnicas y que puede aplicar en sus trabajos para hacer sus productos más poderosos y eficientes.

8. Referencias y lecturas

1. http://today.java.net/pub/a/today/2006/10/19/invoking-assembly-language-from-java.html
2. Java Native Interface: Programmer's guide and specification, http://java.sun.com/docs/books/jni/html/jniTOC.html
3. http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/intro.html#wp725
4. http://java.sun.com/docs/books/jni/html/intro.html#6922
5. http://notepad-plus.sourceforge.net/es/site.htm
6. http://www.masm32.com/masmdl.htm

7. http://java.sun.com/products/archive/j2se/5.0_18/index.html
8. Variables de entorno, http://es.wikipedia.org/wiki/Variable_de_entorno
9. Para conocer el nombrado de los distintos tipos de datos que JNI asigna a los parámetros se puede consultar en http://www.javahispano.org/contenidos/es/jni_java_native_interface/, en la sección *Nomenclatura de tipos de datos*.
10. http://es.wikipedia.org/wiki/FPU

Jorge Ruiz Aquino (jesfre)
jesfre.gy en gmail.com

Actualmente estoy cursando el último grado de Ingeniería en Sistemas Computacionales en la Universidad de Montemorelos, en Nuevo León, México, y trabajo en JWM Solutions como programador JEE. He participado en proyectos JEE en mi anterior trabajo y el actual, entre otros pequeños proyectos J2SE durante la carrera universitaria, en algunos de los cuales he tenido oportunidad de involucrar JNI.