Table of Contents

Table of Contents

LESSON: Java Native Interface (JNI)

author: Edwin van der Elst − edwin@finalist.com
organization: Finalist IT Group − finalist.com

Slides

1. Overview
2. What is JNI
3. When to use JNI
4. When & why NOT to use JNI
5. Calling C/C++ from Java
6. Example − Declare Native Methods
7. Example − Generate .h File
8. Example − Implement Native Function
9. Example − Compile and Link Library (Windows)
10. Example − Compile and Link Library (Linux/GCC)
11. Example − Run
12. Passing Parameters
13. Type Conversion − Primitives Types
14. Type Conversion − Object Types
15. Accessing Strings
16. Calling Java Methods from C++
17. Calling Java Methods from C++ (continued)
18. Calling Java Methods from C++ (continued)
19. Callling Java from C++
20. Finding the Signature
21. Signature Types
22. Memory Management
23. Memory Management − Object References
24. MultiThreading
25. MultiThreading − Native Code Considerations
26. MultiThreading − Synchronization

Labs

1. Simple Native Call
2. Calling Java Method from Native Code

Resources

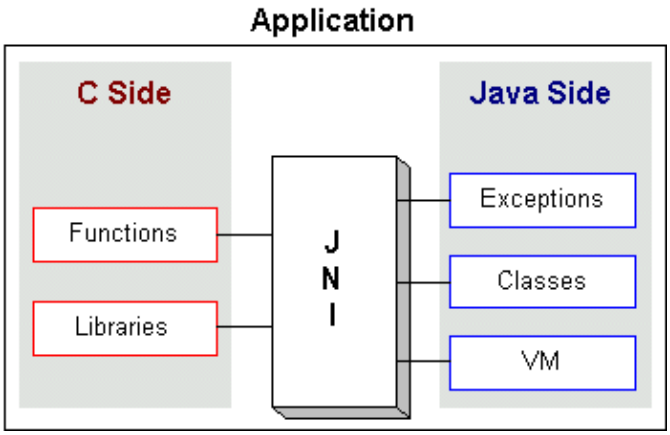
Overview

- What is JNI ?
- When to use JNI, when not to use JNI
- Calling C/C++ from Java
- Type conversions: primitives
- Type conversions: objects, ...
- Memory management
- Calling Java from C/C++
    - ♦ callbacks
    - ♦ starting a JVM from within a C program

What is JNI

A specification/API for calling 'native' code from Java and Java code from native applications ('Invocation API'). JNI is part of the JDK.

Native in practice means C/C++ but could be assembler or other languages.



When to use JNI

- When a program cannot be written in Java alone
- To integrate or reuse legacy applications/libraries
- When platform−specific functionality is not accessible using the Java API
- To optimize a (small) time critical part of an application

When & why NOT to use JNI

- When a program can be written in Java alone
- When platform independence is a must, C code can easily contain platform specific code!

Calling C/C++ from Java

Calling C/C++ functions from Java requires that those functions have a special signature. The following steps are required to create and call C functions:

- Write Java class with native method declarations
- compile class
- use commandline tool ***javah*** to generate C signature
- Implement function in native libary
- write Java code that uses the library

Example − Declare Native Methods

Declare method justSayHello() as native

```
class JniExample1 {
    public native void justSayHello();

}
```

Compile: javac JniExample.java

Example − Generate .h File

javah generates .h file with C signature of the functions that must be implemented in native code.

```
javah −classpath . JniExample1
```

```
 1: /* DO NOT EDIT THIS FILE − it is machine generated */
 2: #include <jni.h>
 3: /* Header for class JniExample1 */
 4:
 5: #ifndef _Included_JniExample1
 6: #define _Included_JniExample1
 7: #ifdef __cplusplus
 8: extern "C" {
 9: #endif
10: /*
11:  * Class:     JniExample1
12:  * Method:    justSayHello
13:  * Signature: ()V
14:  */
15: JNIEXPORT void JNICALL Java_JniExample1_justSayHello
16:   (JNIEnv *, jobject);
17:
18: #ifdef __cplusplus
19: }
20: #endif
21: #endif
```

Example − Implement Native Function

```
 1: #include <jni.h>
 2: #include <stdio.h>
 3:
 4: #include "JniExample1.h"
 5:
 6: JNIEXPORT void JNICALL
 7: Java_JniExample1_justSayHello(JNIEnv *, jobject) {
 8:   printf("Hello there");
 9:   return ;
10: }
11:
12:
13:
```

Example − Compile and Link Library (Windows)

Compile (creates JniExample1.obj)

bcc32 −Ic:\borland\bcc55\include −Ic:\jdk1.3.1\include −IC:\jdk1.3.1\include\win32 −c JniExample1.cpp

Link (creates JniExample1.dll)

ilink32 −Tpd −Lc:\borland\bcc55\lib JniExample1.obj, JniExample1.dll, ,c0s32.obj cw32.lib import32.lib

Result: JniExample1.dll

This example uses a C++ compiler because the C++ syntax is cleaner for JNI.

The Borland C++ compiler is freely downloadable from Borlands website. The GNU GCC compiler can be used for Windows development, but requires modification of jni.h.

Example − Compile and Link Library (Linux/GCC)

```
 1: #!/bin/sh
 2: #
 3: # Sample build for .so for Linux with gcc
 4: # Note that if the library is loaded as "JniExample1"
 5: # through System.loadLibrary("JniExample1");
 6: # the library output file must be called libJniExample1.so
 7: # In addition you must set LD_LIBRARY_PATH to include
 8: # the directory where the .so file is located when running
 9: # java (java.library.path System property).
10: #
11: # $Id: build−gcc.sh,v 1.1 2003/09/10 08:40:43 justb Exp $
12:
13: gcc −I${JAVA_HOME}/include −I${JAVA_HOME}/include/linux\
14:       −o $2 −shared\
15:       $1 −static −lc
```

Example − Run

```
 1: // $Id: JniExample1.java,v 1.3 2003/09/10 10:02:57 justb Exp $
 2: public class JniExample1 {
 3:     // Declare native method.
 4:     public native void justSayHello();
 5:
 6:     static {
 7:        // Loads JniExample1.dll (Windows) or
 8:        // libJniExample1.so (Unixes)
 9:        System.loadLibrary("JniExample1");
10:     }
11:
12:     public static void main(String[] args) {
13:         JniExample1 example=new JniExample1();
14:
15:         // Call native method
16:         example.justSayHello();
17:     }
18: }
```

java −cp . JniExample1

The static initalization block loads the native library when this class is first used. Note that the file extension (.dll) is omitted

Passing Parameters

When passing parameters, types have to be converted between Java and C.

```
public native void repeatThis(String text, int count);
```

Leads to the following C header file:

```
/*
 * Class:     JniExample2
 * Method:    repeatThis
 * Signature: (Ljava/lang/String;I)V
 */
JNIEXPORT void JNICALL Java_JniExample2_repeatThis
  (JNIEnv *, jobject, jstring, jint);
```


The first 2 parameters are the same for all native methods

JNIEnv * , the JNI interface. Exposes various functions to the native application

jobject , the object upon which the native call is performed. (The 'this' reference on the Java side, for static methods, the instance of Class is supplied)

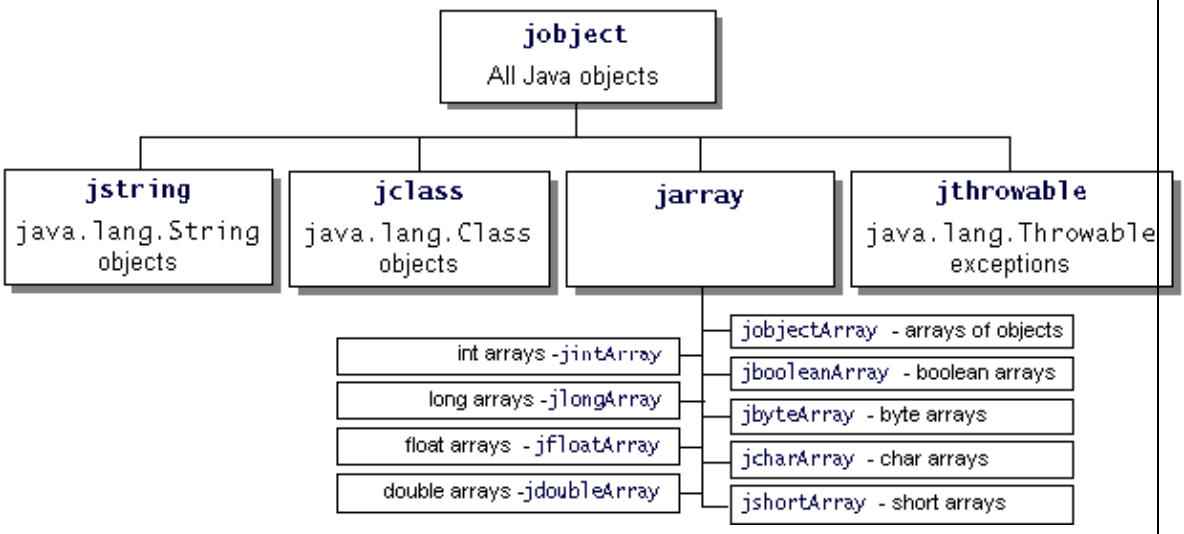
Type Conversion − Primitives Types

Java	*Native*	*Size in bits*
boolean	jboolean	8, unsigned
byte	jbyte	8
char	jchar	16, unsigned
short	jshort	16
int	jint	32
long	jlong	64
float	jfloat	32
double	jdouble	64
void	void	N/A

All primitive types can be used directly in the C code

Type Conversion – Object Types

All object types in Java (including arrays) are passed by reference as a subtype of 'jobject'



Accessing Strings

The Java String type is Unicode (UTF−8). C however, uses null terminated char *

- GetStringUTFChars() − Converts to C style string
- ReleaseStringUTFChars() − Releases allocated memory

```
 1: #include <jni.h>
 2: #include <stdio.h>
 3:
 4: #include "JniExample2.h"
 5:
 6: JNIEXPORT void JNICALL Java_JniExample2_repeatThis (JNIEnv * env,
 7:     jobject o, jstring text, jint count) {
 8:     const char *str = env->GetStringUTFChars(text, 0);
 9:     for (int i=0; i<count; i++) {
10:             printf("%s\n", str);
11:     }
12:     env->ReleaseStringUTFChars(text, str);
13:   return ;
14: }
15:
16:
17:
```


Forgetting to release the allocated C string causes memory leakage, since the Java garbage collector isn't aware of memory allocated in native code.

Calling Java Methods from C++

Calling Java methods from C/C++ code can be useful in the following situations:

- Callbacks: the native code needs to call a method on a supplied java object. Example: calling a getXYZ() method to access a bean property.
- Embedding Java in native programs: the JVM is loaded by the native code. The native code can instantiate Java objects and invoke methods.

Calling Java Methods from C++ (continued)

Calling Java methods looks a lot like reflection in Java. Use the JNIEnv 'type' in C to find the method.

Steps involved:

- Get the 'Class' instance of the object
- Obtain the 'MethodID' of the method
- Call the appropriate CallXYZ method on the JNIEnv instance.

Calling Java Methods from C++ (continued)

Java code:

```
public native void printInfo();

private int number=0;
public String toString() {
     return "JniExample3.number=="+number;
}
```

Objective: call 'toString' from the native 'printInfo'

We will test this with:

```
public static void main(String[] args) {
   JniExample3 example=new JniExample3(5);
   example.printInfo();
}
```

Callling Java from C++

Finding the 'Class' instance

```
JNIEXPORT void JNICALL Java_JniExample3_printInfo
  (JNIEnv *env, jobject o) {
    // Get hold of the class
    jclass clazz=env->GetObjectClass(o);
```

Finding the 'MethodID'

```
jmethodID id=env->GetMethodID(clazz,"toString","()Ljava/lang/String;");
if (id==0) {
    return ;
}
```


A return when methodID==0 will result in a NoSuchMethodError in the Java application

You can cache the MethodID for more calls to the Java method to increase performance. But the id is only valid until the class is unloaded by the VM. Therefore, you should keep a reference to the class (see: Memory Management).

Finding the Signature

- JNI uses signatures based on parameter types and return values
- General form is (argument−types)return−type
- Find signatures with the commandline tool javap

Java Native Interface (JNI)	20

The easiest way to find signatures is the commandline tool 'javap'.
Type: javap −s −p classname

Signature Types

Java type	Signature
boolean	Z
byte	B
char	C
short	S
int	I
long	J
float	F
double	D
a class	L*fully−qualified−class*;
type[]	[type
method	(arg−types)ret−type

The easiest way to find signatures is the commandline tool 'javap'.
Type: javap −s −p classname

Memory Management

Since the Java garbage collector is not aware of any memory allocated within the native code the following applies:

- Native code should ensure proper memory management (new/delete, malloc/free etc)
- References to Java objects within native code should follow JNI conventions (see next)

Java Native Interface (JNI) 22

Memory Management − Object References

In general the Java garbage collector is not aware of references to Java objects passed to native code. *References* protect objects from being garbage collected:

- Local References − valid until the native method returns
- Global References − valid until explicitly freed

All parameters passed to native functions are Local References. A Global Reference can be created from a Local Reference using the following functions:

```
env->NewGlobalRef(jobject local)
env->DeleteGlobalRef(jobject global)
```


Deleting local references is not required since the Garbage Collector will take care of this. env−>DeleteLocalRef(jobject ref) can be used if there is a risk of running out of memory during the execution of a native method.

MultiThreading

When using the JNI consider the following:

- The Java platform is a multithreaded system
- Hence assume that multiple threads may be executing a native method at any given time

Thus native methods must be thread−safe programs

Unless you have knowledge to the contrary, such as knowing that the native method is synchronized, you must assume that there can be multiple threads of control executing a native method at any given time. Native methods therefore must not modify sensitive global variables in unprotected ways. That is, they must share and coordinate their access to variables in certain critical sections of code.

MultiThreading − Native Code Considerations

Within your native (e.g. C/C++) code consider the following: • The JNI interface pointer JNIEnv* is only valid in the current thread • Do not pass local references from one thread to another • Check the (concurrent) use of global variables carefully	
Java Native Interface (JNI)	25

The JNI interface pointer (JNIEnv*) is only valid in the current thread. You must not pass the interface pointer from one thread to another, or cache an interface pointer and use it in multiple threads. The Java Virtual Machine will pass you the same interface pointer in consecutive invocations of a native method from the same thread. However, different threads pass different interface pointers to native methods.

You must not pass local references from one thread to another. In particular, a local reference may become invalid before the other thread has had a chance to use it. You should always convert local references to global references in situations where different threads may be using the same reference to a Java object.

Check the use of global variables carefully. Multiple threads might be accessing these global variables at the same time. Make sure you put in appropriate locks to ensure safety.

MultiThreading − Synchronization

Synchronized blocks can be implemented within Java or the native code • Within Java: using synchronized blocks • Within JNI: using the functions MonitorEnter() and MonitorExit() Unless the native code is fully thread safe, the preferred way is to synchronize within Java.

In Java, you implement synchronized blocks using the synchronized statement. For example:

```
synchronized (obj) {
    ...        /* synchronized block */
    ...
}
```

The Java Virtual Machine guarantees that a thread must acquire the monitor associated with a Java object obj before it can execute the statements in the block. Therefore, at any given time, there can be at most one thread running inside the synchronized block.

Native code can perform equivalent synchronization on objects using the JNI functions MonitorEnter and MonitorExit. For example:

```
...
(*env)->MonitorEnter(env, obj);
...                     /* synchronized block */
(*env)->MonitorExit(env, obj);
...
```

A thread must enter the monitor associated with obj before it can continue its execution. A thread is allowed to enter a monitor multiple times. The monitor contains a counter signaling how many times it has been entered by a given thread. MonitorEnter increments the counter when the thread enters a monitor it has already entered. MonitorExit decrements the counter. Other threads can enter the monitor when the counter reaches zero (0).

RESOURCES

[javasoft−18] JavaSoft ; *Java Tutorial − Trail: Java Native Interface* ; The lessons in this trail show you how to integrate native code with programs written in Java. You will learn how to write native methods. Native methods are methods implemented in another programming language such as C.
http://java.sun.com/docs/books/tutorial/native1.1

[javasoft−19] Sheng Liang ; *The Java Native Interface − Programmer's Guide and Specification* ; This book covers the JavaTM Native Interface (JNI). It will be useful to you if you are interested in any of the following: integrating a Java application with legacy code written in languages such as C or C++, or incorporating a Java virtual machine implementation into an existing application written in languages such as C or C++, or implementing a Java virtual machine, or understanding the technical issues in language interoperability, in particular how to handle features such as garbage collection and multithreading.
http://java.sun.com/docs/books/jni/

[javaworld−2] Tal Liron ; *Enhance your Java application with Java Native Interface (JNI)* ; In this article, Tal Liron presents JNI−based design and implementation techniques that can help make your application competitive on all platforms without compromising cross−platform deployment. He provides a full step−by−step tutorial for adding a Windows−specific desktop indicator feature. The tutorial contains an example of attaching native threads to a Java Virtual Machine, plus many generic JNI tips and tricks that even non−Windows programmers may find useful. Likewise, Tal covers design considerations, not just implementation issues, so you don't have to be a master programmer to appreciate the lessons learned.
http://www.javaworld.com/javaworld/jw−10−1999/jw−10−jni.html

[ringlord−1] Udo Schuermann ; *The JNI HOW−TO* ; As we have ourselves struggled through some of the finer details of using the Java Native Interface (JNI) recently, we thought that sharing our new−found insights might help others understand the details better. The following instructions for accessing native code from Java through the JNI interface apply to a Unix system with gcc. If you are using Microsoft Windows, Apple Macintosh, or another non−Unix operating systems then you can obviously not follow these instructions too blindly..
http://ringlord.com/publications/jni−howto

LABS

1. Simple Native Call

Implement a function in C++ that can be called from Java code. The function is a trivial function that calculates the area of a circle, given its radius. The important aspect of this exercise is not the calculation itself but, rather, the techniques used in communicating with the C++ code. All JNI programs use these techniques to some extent.

Steps:

1. Create a java class that will declare the native method, calculateAreaOfCircle. For this exercise, name the java class MathFuncs and name the shared library MathFuncsImp, implying that this class and shared library could be extended later to support other math−related methods.
2. Create another java class that will be used to test the native method. Name this class Main. It should have a public static void main method from which the native method will be invoked.
3. Compile the java files with javac.

4. Create the C/C++ header file MathFuncs.h with javah.
5. Implement the native method in C++. Name this file MathFuncs.cpp
6. Compile the C++ file and make a shared library (see examples in slides for how to use the Borland compiler and linker).
7. Test the program by running the java interpreter.

2. Calling Java Method from Native Code

In this exercise, we will declare two print methods in a Java class. One print method, which will be named printStringNative, will be a native method implemented in C++. The other method will be named printStringJava and will be implemented in the Java class. Both methods are void methods that take a String as their only argument. The string argument is what will be printed.

The twist in this exercise is that the native method will not simply call the C/C++ function printf or cout, but will instead call the printStringJava method that we will implement in Java.

Steps:

1. Create a java class that will declare the native method, printStringNative. For this exercise, name the java class PrintFuncs and name the shared library PrintFuncsImp. This class should also implement the public method printStringJava which will take a String as a parameter and print it to the screen. PrintFuncs.java can be used as a starting point. (Expand comments of the form "//+")
2. Create another java class that will be used to test the native method. Name this class Main. It should have a public static void main method from which the native method will be invoked. Main.java can be used as a starting point. (Expand comments of the form "//+")
3. Compile the java files with javac.
4. Create the C/C++ header file PrintFuncs.h with javah.
5. Implement the native method in C++. Name this file PrintFuncs.cpp (or PrintFuncs.C, depending on which extension your compiler accepts.) The C++ function should be named printStringNative and it should take a Java string as a parameter. To print the string, the function should call printStringJava that you implemented in PrintFuncs.java. PrintFuncsImp.cpp can be used as a starting point. (Expand comments of the form "//+".)
6. Compile the C++ file and create a shared library.
7. Test the program by running the Java interpreter.