

Java Programming Tutorial

Java Native Interface (JNI)

1. Introduction

At times, it is necessary to use native codes (C/C++) to overcome the memory management and performance constraints in Java. Java supports native codes via the Java Native Interface (JNI).

JNI is difficult, as it involves two languages and runtimes.

I shall assume that you are familiar with:

1. Java.
2. C/C++ and the GCC Compiler (Read "[GCC and Make](#)").
3. (For Windows) Cygwin or MinGW (Read "[How to Setup Cygwin and MinGW](#)").
4. (For IDE) Eclipse C/C++ Development Tool (CDT) (Read "[Eclipse CDT](#)").

2. Getting Started

2.1 JNI with C

Step 1: Write a Java Class that uses C Codes - HelloJNI.java

```

1  public class HelloJNI {
2      static {
3          System.loadLibrary("hello"); // Load native library at runtime
4                                         // hello.dll (Windows) or libhello.so (Unixes)
5      }
6
7      // Declare a native method sayHello() that receives nothing and returns void
8      private native void sayHello();
9
10     // Test Driver
11     public static void main(String[] args) {
12         new HelloJNI().sayHello(); // invoke the native method
13     }
14 }
```

The static initializer invokes `System.loadLibrary()` to load the native library "Hello" (which contains the native method `sayHello()`) during the class loading. It will be mapped to "hello.dll" in Windows; or "libhello.so" in Unixes. This library shall be included in Java's library path (kept in Java system variable `java.library.path`); otherwise, the program will throw a `UnsatisfiedLinkError`. You could include the library into Java Library's path via VM argument `-Djava.library.path=path_to_Lib`.

Next, we declare the method `sayHello()` as a native instance method, via keyword `native`, which denotes that this method is implemented in another language. A native method does not contain a body. The `sayHello()` is contained in the native library loaded.

The `main()` method allocate an instance of `HelloJNI` and invoke the native method `sayHello()`.

Compile the "HelloJNI.java" into "HelloJNI.class".

```
> javac HelloJNI.java
```

Step 2: Create the C/C++ Header file - HelloJNI.h

Run `javah` utility on the class file to create a header file for C/C++ programs:

```
> javah HelloJNI
```

TABLE OF CONTENTS (HIDE)

1. Introduction
2. Getting Started
 - 2.1 JNI with C
 - 2.2 JNI with C/C++ Mixture
 - 2.3 JNI in Package
 - 2.4 JNI in Eclipse
 - 2.5 JNI in NetBeans
3. JNI Basics
4. Passing Arguments and Result b
 - 4.1 Passing Primitives
 - 4.2 Passing Strings
 - 4.3 Passing Array of Primitives
5. Accessing Object's Variables and
 - 5.1 Accessing Object's Instance Variables
 - 5.2 Accessing Class' Static Variables
 - 5.3 Callback Instance Methods and
 - 5.4 Callback Overridden Superclass
6. Creating Objects and Object Array
 - 6.1 Callback the Constructor to Create
 - 6.2 Array of Objects
7. Local and Global References
8. Debugging JNI Programs

The output is `HelloJNI.h` as follows:

```

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

```

The header declares a C function `Java_HelloJNI_sayHello` as follows:

```
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *, jobject);
```

The naming convention for C function is `Java_{package_and_classname}_{function_name}(JNI arguments)`. The dot in package name shall be replaced by underscore.

The arguments:

- `JNIEnv*`: reference to JNI environment, which lets you access all the JNI functions.
- `jobject`: reference to "this" Java object.

We are not using these arguments in the hello-world example, but will be using them later. Ignore the macros `JNIEXPORT` and `JNICALL` for the time being.

The extern "C" is recognized by C++ compiler only. It notifies the C++ compiler that these functions are to be compiled using C's function naming protocol (instead of C++ naming protocol). C and C++ have different function naming protocols as C++ support function overloading and uses a name mangling scheme to differentiate the overloaded functions. Read ["Name Mangling"](#).

Step 3: C Implementation - `HelloJNI.c`

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "HelloJNI.h"
4
5  // Implementation of native method sayHello() of HelloJNI class
6  JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
7      printf("Hello World!\n");
8      return;
9  }

```

Save the C program as `"HelloJNI.c"`.

The header `"jni.h"` is available under the `"<JAVA_HOME>\include"` and `"<JAVA_HOME>\include\win32"` directories, where `<JAVA_HOME>` is your JDK installed directory (e.g., `"c:\program files\java\jdk1.7.0"`).

The C function simply prints the message `"Hello world!"` to the console.

Compile the C program - this depends on the C compiler you used.

For MinGW GCC in Windows

```

> set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_{xx}
    // Define and Set environment variable JAVA_HOME to JDK installed directory
    // I recommend that you set JAVA_HOME permanently, via "Control Panel" => "System" => "Environment Variables"
> echo %JAVA_HOME%
    // In Windows, you can refer a environment variable by adding % prefix and suffix
> gcc -Wl,--add-stdcall-alias -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o hello.dll HelloJNI.c
    // Compile HelloJNI.c into shared library hello.dll

```

The compiler options used are:

- `-Wl`: The `-Wl` to pass linker option `--add-stdcall-alias` to prevent `UnsatisfiedLinkError` (symbols with a stdcall suffix (@nn) will be exported as-is and also with the suffix stripped). (Some people suggested to use `-Wl,--kill-at.`)
- `-I`: for specifying the header files directories. In this case `"jni.h"` (in `"<JAVA_HOME>\include"`) and `"jni_md.h"` (in `"`

<JAVA_HOME>\include\win32"), where <JAVA_HOME> denotes the JDK installed directory. Enclosed the directory in double quotes if it contains spaces.

- -shared: to generate share library.
- -o: for setting the output filename "hello.dll".

You can also compile and link in two steps:

```
// Compile-only with -c flag. Output is HelloJNI.o
> gcc -c -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" HelloJNI.c

// Link into shared library "hello.dll"
> gcc -Wl,--add-stdcall-alias -shared -o hello.dll HelloJNI.o
```

Try nm (which list all the symbols) on the shared library produced to look for the sayHello() function. Take note the GCC added prefix _ and suffix @8 (the number of bytes of parameters). Check for the function name Java_HelloJNI_sayHello with type "T" (defined).

```
> nm hello.dll | grep say
624011d8 T _Java_HelloJNI_sayHello@8
```

For Cygwin GCC in Windows

You need to define the type __int64 as "long long" via option -D __int64="long long".

For gcc-3, include option -mno-cygwin to build DLL files which are not dependent upon the Cygwin DLL.

```
> gcc-3 -D __int64="long long" -mno-cygwin -Wl,--add-stdcall-alias
-I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o hello.dll HelloJNI.c
```

For gcc-4: I still cannot find the correct compiler option (-mno-cygwin is not supported). The Java program hangs!

Step 4: Run the Java Program

```
> java HelloJNI
or
> java -Djava.library.path=. HelloJNI
```

You may need to specify the library path of the "hello.dll" via VM option -Djava.library.path=<path_to_Lib>, as shown above.

2.2 JNI with C/C++ Mixture

Step 1: Write a Java Class that uses Native Codes - HelloJNICpp.java

```
1 public class HelloJNICpp {
2     static {
3         System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes)
4     }
5
6     // Native method declaration
7     private native void sayHello();
8
9     // Test Driver
10    public static void main(String[] args) {
11        new HelloJNICpp().sayHello(); // Invoke native method
12    }
13 }
```

Compile the HelloJNICpp.java into HelloJNICpp.class.

```
> javac HelloJNICpp.java
```

Step 2: Create the C/C++ Header file - HelloJNICpp.h

```
> javah HelloJNICpp
```

The resultant header file "HelloJNICpp.h" declares the native function as:

```
JNIEXPORT void JNICALL Java_HelloJNICpp_sayHello(JNIEnv *, jobject);
```

Step 3: C/C++ Implementation - HelloJNICppImpl.h, HelloJNICppImpl.cpp, and HelloJNICpp.c

We shall implement the program in C++ (in "HelloJNICppImpl.h" and "HelloJNICppImpl.cpp"), but use a C program ("HelloJNICpp.c") to interface with Java.

C++ Header - "HelloJNICppImpl.h"

```
1 #ifndef _HELLO_JNI_CPP_IMPL_H
```

```

2  #define _HELLO_JNI_CPP_IMPL_H
3
4  #ifdef __cplusplus
5      extern "C" {
6  #endif
7      void sayHello ();
8  #ifdef __cplusplus
9      }
10 #endif
11
12 #endif

```

C++ Implementation - "HelloJNICppImpl.cpp"

```

1  #include "HelloJNICppImpl.h"
2  #include <iostream>
3
4  using namespace std;
5
6  void sayHello () {
7      cout << "Hello World from C++!" << endl;
8      return;
9  }

```

C Program interfacing with Java - "HelloJNICpp.c"

```

1  #include <jni.h>
2  #include "HelloJNICpp.h"
3  #include "HelloJNICppImpl.h"
4
5  JNIEXPORT void JNICALL Java_HelloJNICpp_sayHello (JNIEnv *env, jobject thisObj) {
6      sayHello(); // invoke C++ function
7      return;
8  }

```

Compile the C/C++ programs into shared library ("hello.dll" for Windows).

Using MinGW GCC in Windows

```

> set JAVA_HOME=C:\Program Files\Java\jdk1.7.0_{xx}
> g++ -Wl,--add-stdcall-alias -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32"
    -shared -o hello.dll HelloJNICpp.c HelloJNICppImpl.cpp

```

Step 4: Run the Java Program

```

> java HelloJNICpp
or
> java -Djava.library.path=. HelloJNICpp

```

2.3 JNI in Package

For production, all Java classes shall be kept in proper packages, instead of the default no-name package.

Step 1: JNI Program - myjni\HelloJNI.java

```

1  package myjni;
2
3  public class HelloJNI {
4      static {
5          System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes)
6      }
7      // A native method that receives nothing and returns void
8      private native void sayHello();
9
10     public static void main(String[] args) {
11         new HelloJNI().sayHello(); // invoke the native method
12     }
13 }

```

This JNI class is kept in package "myjni" - to be saved as "myjni\HelloJNI.java".

Compile the JNI program:

```

// change directory to package base directory
> javac myjni\HelloJNI.java

```

Step 2: Generate C/C++ Header

If your JNI program is kept in a package, you need to issue fully-qualified name to generate the C/C++ header. You may need to use -classpath option to specify the classpath of the JNI program and -d option to specify the destination directory.

```
> javah --help
.....

// Change directory to package base directory
> javah -d include myjni.HelloJNI
```

In this example, we decided to place the header file under a "include" sub-directory. The output is "include\myjni_HelloJNI.h".

The header file declares a native function:

```
JNIEXPORT void JNICALL Java_myjni_HelloJNI_sayHello(JNIEnv *, jobject);
```

Take note of the native function naming convention: Java_<fully-qualified-name>_methodName, with dots replaced by underscores.

Step 3: C Implementation - HelloJNI.c

```
1  #include <jni.h>
2  #include <stdio.h>
3  #include "include\myjni_HelloJNI.h"
4
5  JNIEXPORT void JNICALL Java_myjni_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
6      printf("Hello World!\n");
7      return;
8  }
```

Compile the C program:

```
> gcc -Wl,--add-stdcall-alias -I<JAVA_HOME>\include -I<JAVA_HOME>\include\win32 -shared -o hello.dll HelloJNI.c
```

You can now run the JNI program:

```
> java myjni.HelloJNI
```

2.4 JNI in Eclipse

Writing JNI under Eclipse is handy for development Android apps with NDK.

You need to install Eclipse and Eclipse CDT (C/C++ Development Tool) Plugin. Read "[Eclipse for C/C++](#)" on how to install CDT.

Step 1: Create a Java Project

Create a new Java project (says HelloJNI), and the following Java class "HelloJNI.java":

```
public class HelloJNI {
    static {
        System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so (Unixes)
    }

    // Declare native method
    private native void sayHello();

    // Test Driver
    public static void main(String[] args) {
        new HelloJNI().sayHello(); // invoke the native method
    }
}
```

Step 2: Convert the Java Project to C/C++ Makefile Project

Right-click on the "HelloJNI" Java project ⇒ New ⇒ Other... ⇒ Convert to a C/C++ Project (Adds C/C++ Nature) ⇒ Next.

The "Convert to a C/C++ Project" dialog appears. In "Project type", select "[Makefile Project](#)" ⇒ In "Toolchains", select "MinGW GCC" ⇒ Finish.

Now, you can run this project as a Java as well as C/C++ project.

Step 3: Generate C/C++ Header File

Create a directory called "jni" under the project to keep all the C/C++ codes, by right-click on the project ⇒ New ⇒ Folder ⇒ In "Folder name", enter "jni".

Create a "makefile" under the "jni" directory, by right-click on the "jni" folder ⇒ new ⇒ File ⇒ In "File name", enter "makefile" ⇒ Enter the following codes. Take note that you need to use tab (instead of spaces) for the indent.

```
# Define a variable for classpath
CLASS_PATH = ../bin

# Define a virtual path for .class in the bin directory
vpath %.class $(CLASS_PATH)

# $* matches the target filename without the extension
HelloJNI.h : HelloJNI.class
    javah -classpath $(CLASS_PATH) $*
```

This makefile create a target "HelloJNI.h", which has a dependency "HelloJNI.class", and invokes the javah utility on HelloJNI.class (under -classpath) to build the target header file.

Right-click on the makefile ⇒ Make Targets ⇒ Create ⇒ In "Target Name", enter "HelloJNI.h".

Run the makefile for the target "HelloJNI.h", by right-click on the makefile ⇒ Make Targets ⇒ Build ⇒ Select the target "HelloJNI.h" ⇒ Build. The header file "HelloJNI.h" shall be generated in the "jni" directory. Refresh (F5) if necessary. The outputs are:

```
make HelloJNI.h
javah -classpath ../bin HelloJNI
```

Read "GCC and Make" for details about makefile.

Alternatively, you could also use the CMD shell to run the make file:

```
// change directory to the directory containing makefile
> make HelloJNI.h
```

You can even use the CMD shell to run the javah:

```
> javah -classpath ../bin HelloJNI
```

Step 4: C Implementation - HelloJNI.c

Create a C program called "HelloJNI.c", by right-click on the "jni" folder ⇒ New ⇒ Source file ⇒ In "Source file", enter "HelloJNI.c". Enter the following codes:

```
#include <jni.h>
#include <stdio.h>
#include "HelloJNI.h"

JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject thisObj) {
    printf("Hello World!\n");
    return;
}
```

Modify the "makefile" as follows to generate the shared library "hello.dll". (Again, use tab to indent the lines.)

```
# Define a variable for classpath
CLASS_PATH = ../bin

# Define a virtual path for .class in the bin directory
vpath %.class $(CLASS_PATH)

all : hello.dll

# $@ matches the target, $< matches the first dependency
hello.dll : HelloJNI.o
    gcc -Wl,--add-stdcall-alias -shared -o $@ $<

# $@ matches the target, $< matches the first dependency
HelloJNI.o : HelloJNI.c HelloJNI.h
    gcc -I"D:\bin\jdk1.7\include" -I"D:\bin\jdk1.7\include\win32" -c $< -o $@

# $* matches the target filename without the extension
HelloJNI.h : HelloJNI.class
    javah -classpath $(CLASS_PATH) $*

clean :
    rm HelloJNI.h HelloJNI.o hello.dll
```

Right-click on the "makefile" ⇒ Make Targets ⇒ Create ⇒ In "Target Name", enter "all". Repeat to create a target "clean".

Run the makefile for the target "all", by right-click on the makefile ⇒ Make Targets ⇒ Build ⇒ Select the target "all" ⇒ Build. The outputs are:

```
make all
javah -classpath ../bin HelloJNI
gcc -I"D:\bin\jdk1.7\include" -I"D:\bin\jdk1.7\include\win32" -c HelloJNI.c -o HelloJNI.o
```

```
gcc -Wl,--add-stdcall-alias -shared -o hello.dll HelloJNI.o
```

The shared library "hello.dll" shall have been created in "jni" directory.

Step 5: Run the Java JNI Program

You can run the Java JNI program HelloJNI. However, you need to provide the library path to the "hello.dll". This can be done via VM argument `-Djava.library.path`. Right-click on the project ⇒ Run As ⇒ Run Configurations ⇒ Select "Java Application" ⇒ In "Main" tab, enter the main class "HelloJNI" ⇒ In "Arguments", "VM Arguments", enter `-Djava.library.path=jni` ⇒ Run.

You shall see the output "Hello World!" displayed on the console.

2.5 JNI in NetBeans

[TODO]

3. JNI Basics

JNI defines the following JNI types in the native system that correspond to Java types:

1. Java Primitives: jint, jbyte, jshort, jlong, jfloat, jdouble, jchar, jboolean for Java Primitive of int, byte, short, long, float, double, char and boolean, respectively.
2. Java Reference Types: jobject for java.lang.Object. It also defines the following *sub-types*:
 - a. jclass for java.lang.Class.
 - b. jstring for java.lang.String.
 - c. jthrowable for java.lang.Throwable.
 - d. jarray for Java array. Java array is a reference type with eight primitive array and one Object array. Hence, there are eight array of primitives jintArray, jbyteArray, jshortArray, jlongArray, jfloatArray, jdoubleArray, jcharArray and jbooleanArray; and one object array jobjectArray.

The native functions receives argument in the above JNI types and returns a value in the JNI type (such as jstring, jintArray). However, native functions operate on their own native types (such as C-string, C's int[]). Hence, there is a need to convert (or transform) between JNI types and the native types.

The native programs:

1. Receive the arguments in JNI type (passed over by the Java program).
2. For reference JNI type, convert or copy the arguments to local native types, e.g., jstring to a C-string, jintArray to C's int[], and so on. Primitive JNI types such as jint and jdouble do not need conversion and can be operated directly.
3. Perform its operations, in local native type.
4. Create the returned object in JNI type, and copy the result into the returned object.
5. Return.

The most confusing and challenging task in JNI programming is the conversion (or transformation) between JNI *reference* types (such as jstring, jobject, jintArray, jobjectArray) and native types (C-string, int[]). The JNI Environment interface provides many functions to do the conversion.

JNI is a C interface, which is not object-oriented. It does not really pass the objects.

[C++ object-oriented interface?!]

4. Passing Arguments and Result between Java & Native Programs

4.1 Passing Primitives

Passing Java primitives is straight forward. A jxxx type is defined in the native system, i.e. jint, jbyte, jshort, jlong, jfloat, jdouble, jchar and jboolean for each of the Java's primitives int, byte, short, long, float, double, char and boolean, respectively.

Java JNI Program: TestJNIPrimitive.java

```
1 public class TestJNIPrimitive {
2     static {
3         System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4     }
5
6     // Declare a native method average() that receives two ints and return a double containing the average
7     private native double average(int n1, int n2);
```

```

8
9 // Test Driver
10 public static void main(String args[]) {
11     System.out.println("In Java, the average is " + new TestJNIPrimitive().average(3, 2));
12 }
13 }

```

This JNI program loads a shared library myjni.dll (Windows) or libmyjni.so (Unixes). It declares a native method average() that receives two int's and returns a double containing the average value of the two int's. The main() method invoke the average().

Compile the Java program into "TestJNIPrimitive.class" and generate the C/C++ header file "TestJNIPrimitive.h":

```

> javac TestJNIPrimitive.java
> javah TestJNIPrimitive // Output is TestJNIPrimitive.h

```

C Implementation - TestJNIPrimitive.c

The header file TestJNIPrimitive.h contains a function declaration Java_TestJNIPrimitive_average() which takes a JNIEnv* (for accessing JNI environment interface), a jobject (for referencing this object), two jint's (Java native method's two arguments) and returns a jdouble (Java native method's return-type).

```
JNIEXPORT jdouble JNICALL Java_TestJNIPrimitive_average(JNIEnv *, jobject, jint, jint);
```

The JNI types jint and jdouble correspond to Java's type int and double, respectively.

The "jni.h" and "win32\jni_mh.h" (which is platform dependent) contains these typedef statements for the eight JNI primitives and an additional jsize.

It is interesting to note that jint is mapped to C's long (which is at least 32 bits), instead of C's int (which could be 16 bits). Hence, it is important to use jint in the C program, instead of simply using int. Cygwin does not support __int64.

```

// In "win\jni_mh.h" - machine header which is machine dependent
typedef long      jint;
typedef __int64   jlong;
typedef signed char jbyte;

// In "jni.h"
typedef unsigned char jboolean;
typedef unsigned short jchar;
typedef short jshort;
typedef float jfloat;
typedef double jdouble;
typedef jint jsize;

```

The implementation TestJNIPrimitive.c is as follows:

```

1 #include <jni.h>
2 #include <stdio.h>
3 #include "TestJNIPrimitive.h"
4
5 JNIEXPORT jdouble JNICALL Java_TestJNIPrimitive_average
6     (JNIEnv *env, jobject thisObj, jint n1, jint n2) {
7     jdouble result;
8     printf("In C, the numbers are %d and %d\n", n1, n2);
9     result = ((jdouble)n1 + n2) / 2.0;
10    // jint is mapped to int, jdouble is mapped to double
11    return result;
12 }

```

Compile the C program into shared library (jni.dll).

```

// MinGW GCC under Windows
> set JAVA_HOME={jdk-installed-directory}
> gcc -Wl,--add-stdcall-alias -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o myjni.dll TestJNIPrimitive.c

```

Now, run the Java Program:

```
> java TestJNIPrimitive
```

C++ Implementation - TestJNIPrimitive.cpp

```

1 #include <jni.h>
2 #include <iostream>
3 #include "TestJNIPrimitive.h"
4 using namespace std;
5
6 JNIEXPORT jdouble JNICALL Java_TestJNIPrimitive_average

```



```

7      (JNIEnv *env, jobject obj, jint n1, jint n2) {
8      jdouble result;
9      cout << "In C++, the numbers are " << n1 << " and " << n2 << endl;
10     result = ((jdouble)n1 + n2) / 2.0;
11     // jint is mapped to int, jdouble is mapped to double
12     return result;
13 }

```

Use g++ (instead of gcc) to compile the C++ program:

```

// MinGW GCC under Windows
> g++ -Wl,--add-stdcall-alias -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o myjni.dll TestJNIPrimitive.cpp

```

4.2 Passing Strings

Java JNI Program: TestJNIString.java

```

1  public class TestJNIString {
2      static {
3          System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4      }
5      // Native method that receives a Java String and return a Java String
6      private native String sayHello(String msg);
7
8      public static void main(String args[]) {
9          String result = new TestJNIString().sayHello("Hello from Java");
10         System.out.println("In Java, the returned string is: " + result);
11     }
12 }

```

This JNI program declares a native method sayHello() that receives a Java String and returns a Java String. The main() method invokes the sayHello().

Compile the Java program and generate the C/C++ header file "TestJNIString.h":

```

> javac TestJNIString.java
> javah TestJNIString

```

C Implementation - TestJNIString.c

The header file TestJNIString.h contains this function declaration:

```
JNIEXPORT jstring JNICALL Java_TestJNIString_sayHello(JNIEnv *, jobject, jstring);
```

JNI defined a jstring type to represent the Java String. The last argument (of JNI type jstring) is the Java String passed into the C program. The return-type is also jstring.

Passing strings is more complicated than passing primitives, as Java's String is an object (reference type), while C-string is a NULL-terminated char array. You need to convert between Java String (represented as JNI jstring) and C-string (char*).

The JNI Environment (accessed via the argument JNIEnv*) provides functions for the conversion:

1. To get a C-string (char*) from JNI string (jstring), invoke method const char* GetStringUTFChars(JNIEnv*, jstring, jboolean*).
2. To get a JNI string (jstring) from a C-string (char*), invoke method jstring NewStringUTF(JNIEnv*, char*).

The C implementation TestJNIString.c is as follows.

1. It receives the JNI string (jstring), convert into a C-string (char*), via GetStringUTFChars().
2. It then performs its intended operations - displays the string received and prompts user for another string to be returned.
3. It converts the returned C-string (char*) to JNI string (jstring), via NewStringUTF(), and return the jstring.

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNIString.h"
4
5  JNIEXPORT jstring JNICALL Java_TestJNIString_sayHello(JNIEnv *env, jobject thisObj, jstring inJNIStr) {
6      // Step 1: Convert the JNI String (jstring) into C-String (char*)
7      const char *inCStr = (*env)->GetStringUTFChars(env, inJNIStr, NULL);
8      if (NULL == inCStr) return NULL;
9
10     // Step 2: Perform its intended operations
11     printf("In C, the received string is: %s\n", inCStr);
12     (*env)->ReleaseStringUTFChars(env, inJNIStr, inCStr); // release resources
13 }

```

```

14     // Prompt user for a C-string
15     char outCStr[128];
16     printf("Enter a String: ");
17     scanf("%s", outCStr);    // not more than 127 characters
18
19     // Step 3: Convert the C-string (char*) into JNI String (jstring) and return
20     return (*env)->NewStringUTF(env, outCStr);
21 }

```

Compile the C program into shared library.

```

// MinGW GCC under Windows
> gcc -Wl,--add-stdcall-alias -I"<JAVA_HOME>\include" -I"<JAVA_HOME>\include\win32" -shared -o myjni.dll TestJNIString.c

```

Now, run the Java Program:

```

> java TestJNIString
In C, the received string is: Hello from Java
Enter a String: test
In Java, the returned string is: test

```

JNI Native String Functions

JNI supports conversion for Unicode (16-bit characters) and UTF-8 (encoded in 1-3 bytes) strings. UTF-8 strings act like null-terminated C-strings (char array), which should be used in C/C++ programs.

The JNI string (jstring) functions are:

```

// UTF-8 String (encoded to 1-3 byte, backward compatible with 7-bit ASCII)
// Can be mapped to null-terminated char-array C-string
const char * GetStringUTFChars(JNIEnv *env, jstring string, jboolean *isCopy);
    // Returns a pointer to an array of bytes representing the string in modified UTF-8 encoding.
void ReleaseStringUTFChars(JNIEnv *env, jstring string, const char *utf);
    // Informs the VM that the native code no longer needs access to utf.
jstring NewStringUTF(JNIEnv *env, const char *bytes);
    // Constructs a new java.lang.String object from an array of characters in modified UTF-8 encoding.
jsize GetStringUTFLength(JNIEnv *env, jstring string);
    // Returns the length in bytes of the modified UTF-8 representation of a string.
void GetStringUTFRegion(JNIEnv *env, jstring str, jsize start, jsize length, char *buf);
    // Translates len number of Unicode characters beginning at offset start into modified UTF-8 encoding
    // and place the result in the given buffer buf.

// Unicode Strings (16-bit character)
const jchar * GetStringChars(JNIEnv *env, jstring string, jboolean *isCopy);
    // Returns a pointer to the array of Unicode characters
void ReleaseStringChars(JNIEnv *env, jstring string, const jchar *chars);
    // Informs the VM that the native code no longer needs access to chars.
jstring NewString(JNIEnv *env, const jchar *unicodeChars, jsize length);
    // Constructs a new java.lang.String object from an array of Unicode characters.
jsize GetStringLength(JNIEnv *env, jstring string);
    // Returns the length (the count of Unicode characters) of a Java string.
void GetStringRegion(JNIEnv *env, jstring str, jsize start, jsize length, jchar *buf);
    // Copies len number of Unicode characters beginning at offset start to the given buffer buf

```

UTF-8 strings or C-strings

The GetStringUTFChars() function can be used to create a new C-string (char*) from the given Java's jstring. The function returns NULL if the memory cannot be allocated. It is always a good practice to check against NULL.

The 3rd parameter isCopy (of jboolean*), which is an "in-out" parameter, will be set to JNI_TRUE if the returned string is a copy of the original java.lang.String instance. It will be set to JNI_FALSE if the returned string is a direct pointer to the original String instance - in this case, the native code shall not modify the contents of the returned string. The JNI runtime will try to return a direct pointer, if possible; otherwise, it returns a copy. Nonetheless, we seldom interested in modifying the underlying string, and often pass a NULL pointer.

Always invoke ReleaseStringUTFChars() whenever you do not need the returned string of GetStringUTFChars() to release the memory and the reference so that it can be garbage-collected.

The NewStringUTF() function create a new JNI string (jstring), with the given C-string.

JDK 1.2 introduces the GetStringUTFRegion(), which copies the jstring (or a portion from start of length) into the "pre-allocated" C's char array. They can be used in place of GetStringUTFChars(). The isCopy is not needed as the C's array is pre-allocated.

JDK 1.2 also introduces the Get/ReleaseStringCritical() functions. Similar to GetStringUTFChars(), it returns a direct pointer if possible; otherwise, it returns a copy. The native method shall not block (for IO or others) between a pair a GetStringCritical() and ReleaseStringCritical() call.

For detailed description, always refer to "Java Native Interface Specification" @

<http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html>.

Unicode String

Instead of `char*`, it uses a `jchar*` to store the Unicode characters.

C++ Implementation - TestJNIString.cpp

```

1  #include <jni.h>
2  #include <iostream>
3  #include <string>
4  #include "TestJNIString.h"
5  using namespace std;
6
7  JNIEXPORT jstring JNICALL Java_TestJNIString_sayHello(JNIEnv *env, jobject thisObj, jstring inJNIStr) {
8      // Step 1: Convert the JNI String (jstring) into C-String (char*)
9      const char *inCStr = env->GetStringUTFChars(inJNIStr, NULL);
10     if (NULL == inCStr) return NULL;
11
12     // Step 2: Perform its intended operations
13     cout << "In C++, the received string is: " << inCStr << endl;
14     env->ReleaseStringUTFChars(inJNIStr, inCStr); // release resources
15
16     // Prompt user for a C++ string
17     string outCppStr;
18     cout << "Enter a String: ";
19     cin >> outCppStr;
20
21     // Step 3: Convert the C++ string to C-string, then to JNI String (jstring) and return
22     return env->NewStringUTF(outCppStr.c_str());
23 }
```

Use `g++` to compile the C++ program:

```

// MinGW GCC under Windows
> g++ -Wl,--add-stdcall-alias -I"<JAVA_HOME>\include" -I"<JAVA_HOME>\include\win32" -shared -o myjni.dll TestJNIString.cpp
```

Take note that C++ native string functions have different syntax from C. In C++, we could use `env->`, instead of `*(env*)->`. Furthermore, there is no need for the `JNIEnv*` argument in the C++ functions.

Also take note that C++ support a string class (under the header `<string>` which is more user-friendly, as well as the legacy C-string (`char` array).

[TODO] Is C++ string class supported directly?

4.3 Passing Array of Primitives

JNI Program - TestJNIPrimitiveArray.java

```

1  public class TestJNIPrimitiveArray {
2      static {
3          System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4      }
5
6      // Declare a native method sumAndAverage() that receives an int[] and
7      // return a double[2] array with [0] as sum and [1] as average
8      private native double[] sumAndAverage(int[] numbers);
9
10     // Test Driver
11     public static void main(String args[]) {
12         int[] numbers = {22, 33, 33};
13         double[] results = new TestJNIPrimitiveArray().sumAndAverage(numbers);
14         System.out.println("In Java, the sum is " + results[0]);
15         System.out.println("In Java, the average is " + results[1]);
16     }
17 }
```

C Implementation - TestJNIPrimitiveArray.c

The header "TestJNIPrimitiveArray.h" contains the following function declaration:

```
JNIEXPORT jdoubleArray JNICALL Java_TestJNIPrimitiveArray_average (JNIEnv *, jobject, jintArray);
```

In Java, array is a *reference type*, similar to a class. There are 9 types of Java arrays, one each of the eight primitives and an array of `java.lang.Object`. JNI defines a type for each of the eight Java primitive arrays, i.e. `jintArray`, `jbyteArray`, `jshortArray`, `jlongArray`,

jfloatArray, jdoubleArray, jcharArray, jbooleanArray for Java's primitive array of int, byte, short, long, float, double, char and boolean, respectively. It also define a jobjectArray for Java's array of Object (to be discussed later).

Again, you need to convert between JNI array and native array, e.g., between jintArray and C's jint[], or jdoubleArray and C's jdouble[]. The JNI Environment interface provides a set of functions for the conversion:

1. To get a C native jint[] from a JNI jintArray, invoke jint* GetIntArrayElements(JNIEnv *env, jintArray a, jboolean *iscopy).
2. To get a JNI jintArray from C native jint[], first, invoke jintArray NewIntArray(JNIEnv *env, jsize len) to allocate, then use void SetIntArrayRegion(JNIEnv *env, jintArray a, jsize start, jsize len, const jint *buf) to copy from the jint[] to jintArray.

There are 8 sets of the above functions, one for each of the eight Java primitives.

The native program is required to:

1. Receive the incoming JNI array (e.g., jintArray), convert to C's native array (e.g., jint[]).
2. Perform its intended operations.
3. Convert the return C's native array (e.g., jdouble[]) to JNI array (e.g., jdoubleArray), and return the JNI array.

The C implementation "TestJNIPrimitiveArray.c" is:

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNIPrimitiveArray.h"
4
5  JNIEXPORT jdoubleArray JNICALL Java_TestJNIPrimitiveArray_sumAndAverage
6      (JNIEnv *env, jobject thisObj, jintArray inJNIArray) {
7      // Step 1: Convert the incoming JNI jintarray to C's jint[]
8      jint *inCArray = (*env)->GetIntArrayElements(env, inJNIArray, NULL);
9      if (NULL == inCArray) return NULL;
10     jsize length = (*env)->GetArrayLength(env, inJNIArray);
11
12     // Step 2: Perform its intended operations
13     jint sum = 0;
14     int i;
15     for (i = 0; i < length; i++) {
16         sum += inCArray[i];
17     }
18     jdouble average = (jdouble)sum / length;
19     (*env)->ReleaseIntArrayElements(env, inJNIArray, inCArray, 0); // release resources
20
21     jdouble outCArray[] = {sum, average};
22
23     // Step 3: Convert the C's Native jdouble[] to JNI jdoublearray, and return
24     jdoubleArray outJNIArray = (*env)->NewDoubleArray(env, 2); // allocate
25     if (NULL == outJNIArray) return NULL;
26     (*env)->SetDoubleArrayRegion(env, outJNIArray, 0, 2, outCArray); // copy
27     return outJNIArray;
28 }
```

JNI Primitive Array Functions

The JNI primitive array (jintArray, jbyteArray, jshortArray, jlongArray, jfloatArray, jdoubleArray, jcharArray and jbooleanArray) functions are:

```

// ArrayType: jintArray, jbyteArray, jshortArray, jlongArray, jfloatArray, jdoubleArray, jcharArray, jbooleanArray
// PrimitiveType: int, byte, short, long, float, double, char, boolean
// NativeType: jint, jbyte, jshort, jlong, jfloat, jdouble, jchar, jboolean
NativeType * Get<PrimitiveType>ArrayElements(JNIEnv *env, ArrayType array, jboolean *isCopy);
void Release<PrimitiveType>ArrayElements(JNIEnv *env, ArrayType array, NativeType *elems, jint mode);
void Get<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array, jsize start, jsize length, NativeType *buffer);
void Set<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array, jsize start, jsize length, const NativeType *buffer);
ArrayType New<PrimitiveType>Array(JNIEnv *env, jsize length);
void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy);
void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode);
```

The GET|Release<PrimitiveType>ArrayElements() can be used to create a new C's native array jxxx[] from the given Java jxxxArray. GET|Set<PrimitiveType>ArrayRegion() can be used to copy a jxxxArray (or a portion from start of length) to and from a pre-allocated C native array jxxx[].

The New<PrimitiveType>Array() can be used to allocate a new jxxxArray of a given size. You can then use the Set<PrimitiveType>ArrayRegion() function to fill its contents from a native array jxxx[].

The Get|ReleasePrimitiveArrayCritical() functions does not allow blocking calls in between the get and release.

5. Accessing Object's Variables and Calling Back Methods

5.1 Accessing Object's Instance Variables

JNI Program - TestJNIInstanceVariable.java

```

1  public class TestJNIInstanceVariable {
2      static {
3          System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4      }
5
6      // Instance variables
7      private int number = 88;
8      private String message = "Hello from Java";
9
10     // Declare a native method that modifies the instance variables
11     private native void modifyInstanceVariable();
12
13     // Test Driver
14     public static void main(String args[]) {
15         TestJNIInstanceVariable test = new TestJNIInstanceVariable();
16         test.modifyInstanceVariable();
17         System.out.println("In Java, int is " + test.number);
18         System.out.println("In Java, String is " + test.message);
19     }
20 }

```

The class contains two private instance variables: a primitive int called number and a String called message. It also declares a native method, which could modify the contents of the instance variables.

C Implementation - TestJNIInstanceVariable.c

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNIInstanceVariable.h"
4
5  JNIEXPORT void JNICALL Java_TestJNIInstanceVariable_modifyInstanceVariable
6      (JNIEnv *env, jobject thisObj) {
7      // Get a reference to this object's class
8      jclass thisClass = (*env)->GetObjectClass(env, thisObj);
9
10     // int
11     // Get the Field ID of the instance variables "number"
12     jfieldID fidNumber = (*env)->GetFieldID(env, thisClass, "number", "I");
13     if (NULL == fidNumber) return;
14
15     // Get the int given the Field ID
16     jint number = (*env)->GetIntField(env, thisObj, fidNumber);
17     printf("In C, the int is %d\n", number);
18
19     // Change the variable
20     number = 99;
21     (*env)->SetIntField(env, thisObj, fidNumber, number);
22
23     // Get the Field ID of the instance variables "message"
24     jfieldID fidMessage = (*env)->GetFieldID(env, thisClass, "message", "Ljava/lang/String;");
25     if (NULL == fidMessage) return;
26
27     // String
28     // Get the object given the Field ID
29     jstring message = (*env)->GetObjectField(env, thisObj, fidMessage);
30
31     // Create a C-string with the JNI String
32     const char *cStr = (*env)->GetStringUTFChars(env, message, NULL);
33     if (NULL == cStr) return;
34
35     printf("In C, the string is %s\n", cStr);
36     (*env)->ReleaseStringUTFChars(env, message, cStr);
37
38     // Create a new C-string and assign to the JNI string
39     message = (*env)->NewStringUTF(env, "Hello from C");
40     if (NULL == message) return;
41
42     // modify the instance variables

```

```

43     (*env)->SetObjectField(env, thisObj, fidMessage, message);
44 }

```

To access the instance variable of an object:

1. Get a reference to this object's class via `GetObjectClass()`.
2. Get the Field ID of the instance variable to be accessed via `GetFieldID()` from the class reference. You need to provide the variable name and its field descriptor (or signature). For a Java class, the field descriptor is in the form of "L<fully-qualified-name>;", with dot replaced by forward slash (/), e.g., the class descriptor for String is "Ljava/lang/String;". For primitives, use "I" for int, "B" for byte, "S" for short, "J" for long, "F" for float, "D" for double, "C" for char, and "Z" for boolean. For arrays, include a prefix "[", e.g., "[Ljava/lang/Object;" for an array of Object; "[I" for an array of int.
3. Based on the Field ID, retrieve the instance variable via `GetObjectField()` or `Get<primitive-type>Field()` function.
4. To update the instance variable, use the `SetObjectField()` or `Set<primitive-type>Field()` function, providing the Field ID.

The JNI functions for accessing instance variable are:

```

jclass GetObjectClass(JNIEnv *env, jobject obj);
// Returns the class of an object.

jfieldID GetFieldID(JNIEnv *env, jclass cls, const char *name, const char *sig);
// Returns the field ID for an instance variable of a class.

NativeType Get<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID);
void Set<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID, NativeType value);
// Get/Set the value of an instance variable of an object
// <type> includes each of the eight primitive types plus Object.

```

5.2 Accessing Class' Static Variables

Accessing static variables is similar to accessing instance variable, except that you use functions such as `GetStaticFieldID()`, `Get|SetStaticObjectField()`, `Get|SetStatic<Primitive-type>Field()`.

JNI Program - TestJNIStaticVariable.java

```

1  public class TestJNIStaticVariable {
2      static {
3          System.loadLibrary("myjni"); // nyjni.dll (Windows) or libmyjni.so (Unixes)
4      }
5
6      // Static variables
7      private static double number = 55.66;
8
9      // Declare a native method that modifies the static variable
10     private native void modifyStaticVariable();
11
12     // Test Driver
13     public static void main(String args[]) {
14         TestJNIStaticVariable test = new TestJNIStaticVariable();
15         test.modifyStaticVariable();
16         System.out.println("In Java, the double is " + number);
17     }
18 }

```

C Implementation - TestJNIStaticVariable.c

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNIStaticVariable.h"
4
5  JNIEXPORT void JNICALL Java_TestJNIStaticVariable_modifyStaticVariable
6      (JNIEnv *env, jobject thisObj) {
7      // Get a reference to this object's class
8      jclass cls = (*env)->GetObjectClass(env, thisObj);
9
10     // Read the int static variable and modify its value
11     jfieldID fidNumber = (*env)->GetStaticFieldID(env, cls, "number", "D");
12     if (NULL == fidNumber) return;
13     jdouble number = (*env)->GetStaticDoubleField(env, cls, fidNumber);
14     printf("In C, the double is %f\n", number);
15     number = 77.88;
16     (*env)->SetStaticDoubleField(env, cls, fidNumber, number);
17 }

```

The JNI functions for accessing static variable are:

```

jfieldID GetStaticFieldID(JNIEnv *env, jclass cls, const char *name, const char *sig);
// Returns the field ID for a static variable of a class.

NativeType GetStatic<type>Field(JNIEnv *env, jclass clazz, jfieldID fieldID);
void SetStatic<type>Field(JNIEnv *env, jclass clazz, jfieldID fieldID, NativeType value);
// Get/Set the value of a static variable of a class.
// <type> includes each of the eight primitive types plus Object.

```

5.3 Callback Instance Methods and Static Methods

You can callback an instance and static methods from the native code.

JNI Program - TestJNICallBackMethod.java

```

1  public class TestJNICallBackMethod {
2      static {
3          System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4      }
5
6      // Declare a native method that calls back the Java methods below
7      private native void nativeMethod();
8
9      // To be called back by the native code
10     private void callback() {
11         System.out.println("In Java");
12     }
13
14     private void callback(String message) {
15         System.out.println("In Java with " + message);
16     }
17
18     private double callbackAverage(int n1, int n2) {
19         return ((double)n1 + n2) / 2.0;
20     }
21
22     // Static method to be called back
23     private static String callbackStatic() {
24         return "From static Java method";
25     }
26
27     // Test Driver
28     public static void main(String args[]) {
29         new TestJNICallBackMethod().nativeMethod();
30     }
31 }

```

This class declares a native method called `nativeMethod()`, and invoke this `nativeMethod()`. The `nativeMethod()`, in turn, calls back the various instance and static methods defined in this class.

C Implementation - TestJNICallBackMethod.c

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNICallBackMethod.h"
4
5  JNIEXPORT void JNICALL Java_TestJNICallBackMethod_nativeMethod
6      (JNIEnv *env, jobject thisObj) {
7
8      // Get a class reference for this object
9      jclass thisClass = (*env)->GetObjectClass(env, thisObj);
10
11     // Get the Method ID for method "callback", which takes no arg and return void
12     jmethodID midCallBack = (*env)->GetMethodID(env, thisClass, "callback", "()V");
13     if (NULL == midCallBack) return;
14     printf("In C, call back Java's callback()\n");
15     // Call back the method (which returns void), baed on the Method ID
16     (*env)->CallVoidMethod(env, thisObj, midCallBack);
17
18     jmethodID midCallBackStr = (*env)->GetMethodID(env, thisClass,
19         "callback", "(Ljava/lang/String;)V");
20     if (NULL == midCallBackStr) return;
21     printf("In C, call back Java's called(String)\n");
22     jstring message = (*env)->NewStringUTF(env, "Hello from C");
23     (*env)->CallVoidMethod(env, thisObj, midCallBackStr, message);
24 }

```



```

25     jmethodID midCallBackAverage = (*env)->GetMethodID(env, thisClass,
26                                     "callbackAverage", "(II)D");
27     if (NULL == midCallBackAverage) return;
28     jdouble average = (*env)->CallDoubleMethod(env, thisObj, midCallBackAverage, 2, 3);
29     printf("In C, the average is %f\n", average);
30
31     jmethodID midCallBackStatic = (*env)->GetStaticMethodID(env, thisClass,
32                                                              "callbackStatic", "()Ljava/lang/String;");
33     if (NULL == midCallBackStatic) return;
34     jstring resultJNIString = (*env)->CallStaticObjectMethod(env, thisClass, midCallBackStatic);
35     const char *resultCStr = (*env)->GetStringUTFChars(env, resultJNIString, NULL);
36     if (NULL == resultCStr) return;
37     printf("In C, the returned string is %s\n", resultCStr);
38     (*env)->ReleaseStringUTFChars(env, resultJNIString, resultCStr);
39 }

```

To call back an instance method from the native code:

1. Get a reference to this object's class via `GetObjectClass()`.
2. From the class reference, get the Method ID via `GetMethodID()`. You need to provide the method name and the signature. The signature is in the form "*(parameters)return-type*". You can list the method signature for a Java program via `javap` utility (Class File Disassembler) with `-s` (print signature) and `-p` (show private members):

```

> javap --help
> javap -s -p TestJNICallBackMethod
.....
private void callback();
    Signature: ()V

private void callback(java.lang.String);
    Signature: (Ljava/lang/String;)V

private double callbackAverage(int, int);
    Signature: (II)D

private static java.lang.String callbackStatic();
    Signature: ()Ljava/lang/String;
.....

```

3. Based on the Method ID, you could invoke `Call<Primitive-type>Method()` or `CallVoidMethod()` or `CallObjectMethod()`, where the return-type is *<Primitive-type>*, void and Object, respectively. Append the argument, if any, before the argument list. For non-void return-type, the method returns a value.

To callback a static method, use `GetMethodID()`, `CallStatic<Primitive-type>Method()`, `CallStaticVoidMethod()` or `CallStaticObjectMethod()`.

The JNI functions for calling back instance method and static method are:

```

jmethodID GetMethodID(JNIEnv *env, jclass cls, const char *name, const char *sig);
    // Returns the method ID for an instance method of a class or interface.

NativeType Call<type>Method(JNIEnv *env, jobject obj, jmethodID methodID, ...);
NativeType Call<type>MethodA(JNIEnv *env, jobject obj, jmethodID methodID, const jvalue *args);
NativeType Call<type>MethodV(JNIEnv *env, jobject obj, jmethodID methodID, va_list args);
    // Invoke an instance method of the object.
    // The <type> includes each of the eight primitive and Object.

jmethodID GetStaticMethodID(JNIEnv *env, jclass cls, const char *name, const char *sig);
    // Returns the method ID for an instance method of a class or interface.

NativeType CallStatic<type>Method(JNIEnv *env, jclass clazz, jmethodID methodID, ...);
NativeType CallStatic<type>MethodA(JNIEnv *env, jclass clazz, jmethodID methodID, const jvalue *args);
NativeType CallStatic<type>MethodV(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);
    // Invoke an instance method of the object.
    // The <type> includes each of the eight primitive and Object.

```

5.4 Callback Overridden Superclass' Instance Method

JNI provides a set of `CallNonvirtual<Type>Method()` functions to invoke superclass' instance methods which has been overridden in this class (similar to a super.*methodName()* call inside a Java subclass):

1. Get the Method ID, via `GetMethodID()`.
2. Based on the Method ID, invoke one of the `CallNonvirtual<Type>Method()`, with the object, superclass, and arguments.

The JNI function for calling the overridden superclass' instance method are:

```

NativeType CallNonvirtual<type>Method(JNIEnv *env, jobject obj, jclass cls, jmethodID methodID, ...);

```



```
NativeType CallNonvirtual<type>MethodA(JNIEnv *env, jobject obj, jclass cls, jmethodID methodID, const jvalue *args);
NativeType CallNonvirtual<type>MethodV(JNIEnv *env, jobject obj, jclass cls, jmethodID methodID, va_list args);
```

6. Creating Objects and Object Arrays

You can construct `jobject` and `jobjectArray` inside the native code, via `NewObject()` and `newObjectArray()` functions, and pass them back to the Java program.

6.1 Callback the Constructor to Create a New Java Object in the Native Code

Callback the constructor is similar to calling back method. First at first, Get the Method ID of the constructor by passing "<init>" as the method name and "V" as the return-type. You can then use methods like `NewObject()` to call the constructor to create a new java object.

JNI Program - TestJavaConstructor.java

```
1 public class TestJNIConstructor {
2     static {
3         System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4     }
5
6     // Native method that calls back the constructor and return the constructed object.
7     // Return an Integer object with the given int.
8     private native Integer getIntegerObject(int number);
9
10    public static void main(String args[]) {
11        TestJNIConstructor obj = new TestJNIConstructor();
12        System.out.println("In Java, the number is :" + obj.getIntegerObject(9999));
13    }
14 }
```

This class declares a native method `getIntegerObject()`. The native code shall create and return an `Integer` object, based on the argument given.

C Implementation - TestJavaConstructor.c

```
1 #include <jni.h>
2 #include <stdio.h>
3 #include "TestJNIConstructor.h"
4
5 JNIEXPORT jobject JNICALL Java_TestJNIConstructor_getIntegerObject
6     (JNIEnv *env, jobject thisObj, jint number) {
7     // Get a class reference for java.lang.Integer
8     jclass cls = (*env)->FindClass(env, "java/lang/Integer");
9
10    // Get the Method ID of the constructor which takes an int
11    jmethodID midInit = (*env)->GetMethodID(env, cls, "<init>", "(I)V");
12    if (NULL == midInit) return NULL;
13    // Call back constructor to allocate a new instance, with an int argument
14    jobject newObj = (*env)->NewObject(env, cls, midInit, number);
15
16    // Try running the toString() on this newly create object
17    jmethodID midToString = (*env)->GetMethodID(env, cls, "toString", "()Ljava/lang/String;");
18    if (NULL == midToString) return NULL;
19    jstring resultStr = (*env)->CallObjectMethod(env, newObj, midToString);
20    const char *resultCStr = (*env)->GetStringUTFChars(env, resultStr, NULL);
21    printf("In C: the number is %s\n", resultCStr);
22
23    return newObj;
24 }
```

The JNI functions for creating object (`jobject`) are:

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

jobject NewObject(JNIEnv *env, jclass cls, jmethodID methodID, ...);
jobject NewObjectA(JNIEnv *env, jclass cls, jmethodID methodID, const jvalue *args);
jobject NewObjectV(JNIEnv *env, jclass cls, jmethodID methodID, va_list args);
    // Constructs a new Java object. The method ID indicates which constructor method to invoke

jobject AllocObject(JNIEnv *env, jclass cls);
    // Allocates a new Java object without invoking any of the constructors for the object.
```

6.2 Array of Objects

JNI Program - TestJNIObjectArray.java

```

1  import java.util.ArrayList;
2
3  public class TestJNIObjectArray {
4      static {
5          System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
6      }
7      // Native method that receives an Integer[] and
8      // returns a Double[2] with [0] as sum and [1] as average
9      private native Double[] sumAndAverage(Integer[] numbers);
10
11     public static void main(String args[]) {
12         Integer[] numbers = {11, 22, 32}; // auto-box
13         Double[] results = new TestJNIObjectArray().sumAndAverage(numbers);
14         System.out.println("In Java, the sum is " + results[0]); // auto-unbox
15         System.out.println("In Java, the average is " + results[1]);
16     }
17 }

```

For illustration, this class declares a native method that takes an array of Integer, compute their sum and average, and returns as an array of Double. Take note the arrays of objects are pass into and out of the native method.

C Implementation - TestJNIObjectArray.c

```

1  #include <jni.h>
2  #include <stdio.h>
3  #include "TestJNIObjectArray.h"
4
5  JNIEXPORT jobjectArray JNICALL Java_TestJNIObjectArray_sumAndAverage
6      (JNIEnv *env, jobject thisObj, jobjectArray inJNIArray) {
7      // Get a class reference for java.lang.Integer
8      jclass classInteger = (*env)->FindClass(env, "java/lang/Integer");
9      // Use Integer.intValue() to retrieve the int
10     jmethodID midIntValue = (*env)->GetMethodID(env, classInteger, "intValue", "()I");
11     if (NULL == midIntValue) return NULL;
12
13     // Get the value of each Integer object in the array
14     jsize length = (*env)->GetArrayLength(env, inJNIArray);
15     jint sum = 0;
16     int i;
17     for (i = 0; i < length; i++) {
18         jobject objInteger = (*env)->GetObjectArrayElement(env, inJNIArray, i);
19         if (NULL == objInteger) return NULL;
20         jint value = (*env)->CallIntMethod(env, objInteger, midIntValue);
21         sum += value;
22     }
23     double average = (double)sum / length;
24     printf("In C, the sum is %d\n", sum);
25     printf("In C, the average is %f\n", average);
26
27     // Get a class reference for java.lang.Double
28     jclass classDouble = (*env)->FindClass(env, "java/lang/Double");
29
30     // Allocate a jobjectArray of 2 java.lang.Double
31     jobjectArray outJNIArray = (*env)->NewObjectArray(env, 2, classDouble, NULL);
32
33     // Construct 2 Double objects by calling the constructor
34     jmethodID midDoubleInit = (*env)->GetMethodID(env, classDouble, "<init>", "(D)V");
35     if (NULL == midDoubleInit) return NULL;
36     jobject objSum = (*env)->NewObject(env, classDouble, midDoubleInit, (double)sum);
37     jobject objAve = (*env)->NewObject(env, classDouble, midDoubleInit, average);
38     // Set to the jobjectArray
39     (*env)->SetObjectArrayElement(env, outJNIArray, 0, objSum);
40     (*env)->SetObjectArrayElement(env, outJNIArray, 1, objAve);
41
42     return outJNIArray;
43 }

```

Unlike primitive array which can be processed in bulk, for object array, you need to use the Get|SetObjectArrayElement() to process each of the elements.

The JNI functions for creating and manipulating object array (jobjectArray) are:

```
jobjectArray NewObjectArray(JNIEnv *env, jsize length, jclass elementClass, jobject initialElement);
// Constructs a new array holding objects in class elementClass.
// All elements are initially set to initialElement.

jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index);
// Returns an element of an Object array.

void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value);
// Sets an element of an Object array.
```

7. Local and Global References

Managing references is critical in writing efficient programs. For example, we often use `FindClass()`, `GetMethodID()`, `GetFieldID()` to retrieve a `jclass`, `jmethodID` and `jfieldID` inside native functions. Instead of performing repeated calls, the values should be obtained once and cached for subsequent usage, to eliminate the overheads.

The JNI divides object references (for `jobject`) used by the native code into two categories: local and global references:

1. A *local reference* is created within the native method, and freed once the method exits. It is valid for the duration of a native method. You can also use JNI function `DeleteLocalRef()` to invalidate a local reference explicitly, so that it is available for garbage collection intermediately. Objects are passed to native methods as local references. All Java objects (`jobject`) returned by JNI functions are local references.
2. A *global reference* remains until it is explicitly freed by the programmer, via the `DeleteGlobalRef()` JNI function. You can create a new global reference from a local reference via JNI function `NewGlobalRef()`.

Example

```
1 public class TestJNIReference {
2     static {
3         System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so (Unixes)
4     }
5
6     // A native method that returns a java.lang.Integer with the given int.
7     private native Integer getIntegerObject(int number);
8
9     // Another native method that also returns a java.lang.Integer with the given int.
10    private native Integer anotherGetIntegerObject(int number);
11
12    public static void main(String args[]) {
13        TestJNIReference test = new TestJNIReference();
14        System.out.println(test.getIntegerObject(1));
15        System.out.println(test.getIntegerObject(2));
16        System.out.println(test.anotherGetIntegerObject(11));
17        System.out.println(test.anotherGetIntegerObject(12));
18        System.out.println(test.getIntegerObject(3));
19        System.out.println(test.anotherGetIntegerObject(13));
20    }
21 }
```

The above JNI program declares two native methods. Both of them create and return a `java.lang.Integer` object.

In the C implementation, we need to get a class reference for `java.lang.Integer`, via `FindClass()`. We then find the method ID for the constructor of `Integer`, and invoke the constructor. However, we wish to cache both the class reference and method ID, to be used for repeated invocation.

The following C implementation does not work!

```
1 #include <jni.h>
2 #include <stdio.h>
3 #include "TestJNIReference.h"
4
5 // Global Reference to the Java class "java.lang.Integer"
6 static jclass classInteger;
7 static jmethodID midIntegerInit;
8
9 jobject getInteger(JNIEnv *env, jobject thisObj, jint number) {
10
11     // Get a class reference for java.lang.Integer if missing
12     if (NULL == classInteger) {
13         printf("Find java.lang.Integer\n");
14         classInteger = (*env)->FindClass(env, "java/lang/Integer");
15     }
```

```

16     if (NULL == classInteger) return NULL;
17
18     // Get the Method ID of the Integer's constructor if missing
19     if (NULL == midIntegerInit) {
20         printf("Get Method ID for java.lang.Integer's constructor\n");
21         midIntegerInit = (*env)->GetMethodID(env, classInteger, "<init>", "(I)V");
22     }
23     if (NULL == midIntegerInit) return NULL;
24
25     // Call back constructor to allocate a new instance, with an int argument
26     jobject newObj = (*env)->NewObject(env, classInteger, midIntegerInit, number);
27     printf("In C, constructed java.lang.Integer with number %d\n", number);
28     return newObj;
29 }
30
31 JNIEXPORT jobject JNICALL Java_TestJNIReference_getIntegerObject
32     (JNIEnv *env, jobject thisObj, jint number) {
33     return getInteger(env, thisObj, number);
34 }
35
36 JNIEXPORT jobject JNICALL Java_TestJNIReference_anotherGetIntegerObject
37     (JNIEnv *env, jobject thisObj, jint number) {
38     return getInteger(env, thisObj, number);
39 }

```

In the above program, we invoke `FindClass()` to find the class reference for `java.lang.Integer`, and saved it in a global static variable. Nonetheless, in the next invocation, this reference is no longer valid (and not `NULL`). This is because `FindClass()` returns a local reference, which is invalidated once the method exits.

To overcome the problem, we need to create a global reference from the local reference returned by `FindClass()`. We can then free the local reference. The revised code is as follows:

```

// Get a class reference for java.lang.Integer if missing
if (NULL == classInteger) {
    printf("Find java.lang.Integer\n");
    // FindClass returns a local reference
    jclass classIntegerLocal = (*env)->FindClass(env, "java/lang/Integer");
    // Create a global reference from the local reference
    classInteger = (*env)->NewGlobalRef(env, classIntegerLocal);
    // No longer need the local reference, free it!
    (*env)->DeleteLocalRef(env, classIntegerLocal);
}

```

Take note that `jmethodID` and `jfieldID` are not `jobject`, and cannot create global reference.

8. Debugging JNI Programs

[TODO]

REFERENCES & RESOURCES

1. Java Native Interface Specification @ <http://docs.oracle.com/javase/7/docs/technotes/guides/jni/index.html>.
2. Wiki "Java Native Interface" @ http://en.wikipedia.org/wiki/Java_Native_Interface.
3. Liang, "The Java Native Interface - Programmer's Guide and Specification", Addison Wesley, 1999, available online @ <http://java.sun.com/docs/books/jni/html/jniTOC.html>.
4. JNI Tips @ <http://developer.android.com/guide/practices/jni.html>.

Latest version tested: JDK 1.7.0, MinGW GCC 4.6.2, Eclipse 4.2 (Juno)
Last modified: February, 2014