Java Programming Tutorial

www3.ntu.edu.sg/home/ehchua/programming/java/javanativeinterface.html

yet another insignificant programming notes... | HOME

TABLE OF CONTENTS (HIDE)

Java Native Interface (JNI)

1. Introduction

At times, it is necessary to use native (non-Java) codes (e.g., 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").

2. Getting Started

2.1 JNI with C

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

```
public class HelloJNI {  // Save as HelloJNI.java
   static {
      System.loadLibrary("hello"); // Load native library hello.dll (Windows)
or libhello.so (Unixes)
                                   //  at runtime
                                   // This library contains a native method
called sayHello()
   }

   // Declare an instance native method sayHello() which receives no parameter
and returns void
   private native void sayHello();

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

The *static initializer* invokes `System.loadLibrary()` to load the native library "`hello`" (which contains a native method called `sayHello()`) during the class loading. It will be mapped to "`hello.dll`" in Windows; or "`libhello.so`" in Unixes/Mac OS X. This library shall be included in Java's library path (kept in *Java system variable* `java.library.path`). You could include the library into Java's library path via VM argument `-Djava.library.path=`*`/path/to/lib`*. The program will throw a `UnsatisfiedLinkError` if the library cannot be found in runtime.

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()` shall be found in the native library loaded.

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

Step 2: Compile the Java Program HelloJNI.java & Generate the C/C++ Header File HelloJNI.h

Starting from JDK 8, you should use "`javac -h`" to compile the Java program AND generate C/C++ header file called `HelloJNI.h` as follows:

```
> javac -h . HelloJNI.java
```

The "`-h dir`" option generates C/C++ header and places it in the directory specified (in the above example, `'.'` for the current directory).

Before JDK 8, you need to compile the Java program using `javac` and generate C/C++ header using a dedicated `javah` utility, as follows. The `javah` utility is no longer available in JDK 10.

```
> javac HelloJNI.java
> javah HelloJNI
```

Inspect the header file `HelloJNI.h`:

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

#ifndef _Included_HelloJNI
#define _Included_HelloJNI
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloJNI
 * Method:    sayHello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *,
jobject);

#ifdef __cplusplus
}
#endif
#endif
```

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

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

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

The arguments are:

- `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 this 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: Implementing the C Program HelloJNI.c

```
// Save as "HelloJNI.c"
#include <jni.h>        // JNI header provided by JDK
#include <stdio.h>      // C Standard IO Header
#include "HelloJNI.h"   // Generated

// Implementation of the native method sayHello()
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject
thisObj) {
   printf("Hello World!\n");
   return;
}
```

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

The JNI header "`jni.h`" provided by JDK is available under the "`<JAVA_HOME>\include`" and "`<JAVA_HOME>\include\win32`" (for Windows) or "`<JAVA_HOME>\include\linux`" (for Ubuntu) [Check Mac OS X] directories, where `<JAVA_HOME>` is your JDK installed directory (e.g., "`c:\program files\java\jdk10.0.x`" for Windows).

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

Step 4: Compile the C program HelloJNI.c

Finding the right compiler for your operating platform (Windows, Mac OS X, Ubuntu), for your JDK (32-bit, 64-bit), and figuring out the correct compiler options is the hardest part to get the JNI working!!!

(Windows) 64-bit JDK

We are going to use Cygwin. You need to take note of the followings for Windows:

- Windows/Intel uses these instruction sets: x86 is a 32-bit instruction set; i868 is an enhanced version of x86 (also 32-bit); x86_64 (or amd64) is a 64-bit instruction set.
- A 32-bit compiler can run on 32-bit or 64-bit (backward compatible) Windows, but 64-bit compiler can only run on 64-bit Windows.
- A 64-bit compiler could produce target of 32-bit or 64-bit.
- If you use Cygwin's GCC, the target could be native Windows or Cygwin. If the target is native Windows, the code can be distributed and run under Windows. However, if the target is Cygwin, to distribute, you need to distribute Cygwin runtime environment (`cygwin1.dll`). This is because Cygwin is a Unix emulator under Windows.
- The above explains for the many versions of GCC under Cygwin.

For 64-bit JDK, you need to find a compiler that produces target of 64-bit native Windows. This is provided by MinGW-W64. You can install MinGW-W64 under Cygwin, by selecting packages "`mingw64-x86_64-gcc-core`" (C compiler) and "`mingw64-x86_64-gcc-g++`" (C++ compiler). The executables are `"x86_64-w64-mingw32-gcc`" (C Compiler) and `"x86_64-w64-mingw32-g++`" (C++ Compiler), respectively.

First, set the environment variable `JAVA_HOME` to point the JDK installed directory (e.g., "`c:\program files\java\jdk10.0.x`"). Follow the steps HERE.

Next, use the following commands to compile `HelloJNI.c` into `hello.dll`. In Windows, we reference the environment variable `JAVA_HOME` as `%JAVA_HOME%` in the command.

```
> x86_64-w64-mingw32-gcc -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -
shared -o hello.dll HelloJNI.c
```

The compiler options used are:

- `-IheaderDir`: for specifying the header directory. In this case "`jni.h`" (in "`%JAVA_HOME%\include`") and "`jni_md.h`" (in "`%JAVA_HOME%\include\win32`"), where `JAVA_HOME` is an environment variable set to the JDK installed directory.
- `-shared`: to generate share library.
- `-o outputFilename`: for setting the output filename "`hello.dll`".

You can also compile and link in two steps:

```
// Compile-only "HelloJNI.c" with -c flag. Output is "HElloJNI.o"
> x86_64-w64-mingw32-gcc -c -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32"
HelloJNI.c

// Link "HelloJNO.o" into shared library "hello.dll"
> x86_64-w64-mingw32-gcc -shared -o hello.dll HelloJNI.o
```

You need check the resultant file type via the "`file`" utility, which indicates "`Hello.dll`" is a 64-bit (x86_64) native Windows DLL.

```
> file hello.dll
hello.dll: PE32+ executable (DLL) (console) x86-64, for MS Windows
```

Try `nm`, which lists all the symbols in the shared library and look for the `sayHello()` function. Check for the function name `Java_HelloJNI_sayHello` with type `"T"` (defined).

```
> nm hello.dll | grep say
00000000624014a0 T Java_HelloJNI_sayHello
```

(Windows) 32-bit JDK [Obsolete?]

For 32-bit JDK, you need to find a 32/64-bit compiler that produces target of 32-bit native Windows. This is provided by MinGW-W64 (and the older MinGW). You can install MinGW-W64 under Cygwin, by selecting packages "`mingw64-i686-gcc-core`" (C compiler) and "`mingw64-i686-gcc-g++`" (C++ compiler). The executables are `"i886-w64-mingw32-gcc`" (C Compiler) and `"i686-w64-mingw32-g++`" (C++ Compiler), respectively.

First, set the environment variable `JAVA_HOME` to point the JDK installed directory (e.g., "`c:\program files\java\jdk9.0.x`"). Follow the steps HERE.

Next, use the following command to compile `HelloJNI.c` into `hello.dll`:

```
> i886-w64-mingw32-gcc -Wl,--add-stdcall-alias -I"%JAVA_HOME%\include" -
I"%JAVA_HOME%\include\win32" -shared -o hello.dll HelloJNI.c
```

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%` is an environment variable set to the JDK installed directory.
- `-shared`: to generate share library.
- `-o`: for setting the output filename "`hello.dll`".
- `-D __int64="long long"`: define the type (add this option in front if error "unknown type name '__int64'")

(Ubuntu) 64-bit JDK

1. Set environment variable `JAVA_HOME` to point to the JDK installed directory (which shall contains the `include` subdirectory to be used in the next step):

   ```
   $ export JAVA_HOME=/your/java/installed/dir
   $ echo $JAVA_HOME
   ```

2. Compile the C program `HelloJNI.c` into share module `libhello.so` using `gcc`, which is included in all Unixes:

   ```
   $ gcc -fPIC -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o
   libhello.so HelloJNI.c
   ```

3. Run the Java Program:

   ```
   $ java -Djava.library.path=. HelloJNI
   ```

(Mac OS X) 64-Bit JDK

1. Set environment variable `JAVA_HOME` to point to the JDK installed directory (which shall contains the `include` subdirectory to be used in the next step):

```
$ export JAVA_HOME=/your/java/installed/dir
   // for my machine @
/Library/Java/JavaVirtualMachines/jdk1.8.0_xx.jdk/Contents/Home
$ echo $JAVA_HOME
```

2. Compile the C program `HelloJNI.c` into dynamic share module `libhello.dylib` using `gcc`, which is included in all Unixes/Mac OS:

```
$ gcc -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/darwin" -dynamiclib -o
libhello.dylib HelloJNI.c
```

3. Run the Java Program:

```
$ java -Djava.library.path=. HelloJNI
```

Step 4: Run the Java Program

```
> java HelloJNI
```

You may need to explicitly specify the Java library path of the "`hello.dll`" (Windows), "`libHello.so`" (Unixes), "`libhello.dylib`" (Mac OS X) via VM option `-Djava.library.path=/path/to/lib`, as below. In this example, the native library is kept in the current directory `'.'`.

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

2.2 JNI with C++

Instead of a C program, we can use a C++ program (called `HelloJNI.cpp`) for the above example.

```
// Save as "HelloJNI.cpp"
#include <jni.h>       // JNI header provided by JDK
#include <iostream>    // C++ standard IO header
#include "HelloJNI.h"  // Generated
using namespace std;

// Implementation of the native method sayHello()
JNIEXPORT void JNICALL Java_HelloJNI_sayHello(JNIEnv *env, jobject
thisObj) {
        cout << "Hello World from C++!" << endl;
   return;
}
```

Compile the C++ programs into shared library as follows. See "JNI with C" section for explanation.

(Windows) 64-bit JDK

On Cygwin, you need to install `mingw64-x86-gcc-g++` package.

```
> x86_64-w64-mingw32-g++ -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -
shared -o hello.dll HelloJNI.cpp
```

(Ubuntu) 64-bit JDK

```
$ g++ -fPIC -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o
libhello.so HelloJNI.cpp
```

(Mac OS X) 64-bit JDK

[TODO]

Run the Java Program

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

Notes: If you encounter "`java.lang.UnsatisfiedLinkError: hello.dll: Can't find dependent libraries`", you need to find a "DLL dependency walker" to track down the dependent libraries. Search for the libraries (under Cygwin) and include the libraries in the environment variable `PATH`. In my case, the dependent library is "`libstdc++-6.dll`" located at "`cygwin64\usr\x86_64-w64-mingw32\sys-root\mingw\bin`".

2.3 JNI with C/C++ Mixture

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

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

   // Native method declaration
   private native void sayHello();

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

Step 2: Compile the Java Program & Generate the C/C++ Header file HelloJNICpp.h

```
javac -h . 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`"

```
#ifndef
_HELLO_JNI_CPP_IMPL_H
#define
_HELLO_JNI_CPP_IMPL_H

#ifdef __cplusplus
        extern "C" {
#endif
        void sayHello
();
#ifdef __cplusplus
        }
#endif

#endif
```

C++ Implementation - "HelloJNICppImpl.cpp"

```
#include "HelloJNICppImpl.h"
#include  <iostream>

using namespace std;

void sayHello () {
    cout << "Hello World from C++!" <<
endl;
    return;
}
```

C Program interfacing with Java - "HelloJNICpp.c"

```
#include <jni.h>
#include "HelloJNICpp.h"
#include "HelloJNICppImpl.h"

JNIEXPORT void JNICALL Java_HelloJNICpp_sayHello (JNIEnv *env, jobject
thisObj) {
    sayHello();  // invoke C++ function
    return;
}
```

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

(Windows) 64-bit JDK

On Cygwin, you need to install `mingw64-x86-gcc-g++` package.

```
> x86_64-w64-mingw32-g++ -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -
shared -o hello.dll HelloJNICpp.c HelloJNICppImpl.cpp
```

(Ubuntu) 64-bit JDK

```
$ g++ -fPIC -I"$JAVA_HOME/include" -I"$JAVA_HOME/include/linux" -shared -o
libhello.so HelloJNICpp.c HelloJNICppImpl.cpp
```

Step 4: Run the Java Program

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

Notes: If you encounter "`java.lang.UnsatisfiedLinkError: hello.dll: Can't find dependent libraries`", you need to find a DLL dependency walker to track down the dependent libraries. Search for the libraries (under Cygwin) and include the libraries in the `PATH`. In my case, the dependent library is "`libstdc++-6.dll`" located at "`cygwin64\usr\x86_64-w64-mingw32\sys-root\mingw\bin`".

2.4 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

```
package myjni;

public class HelloJNI {
   static {
      System.loadLibrary("hello"); // hello.dll (Windows) or libhello.so
(Unixes)
   }
   // A native method that receives nothing and returns void
   private native void sayHello();

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

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

Step 2: Compile the JNI program & Generate C/C++ Header

```
// change directory to package base directory
> javac -h include myjni\HelloJNI
```

The output of compilation is "`myjni\HelloJNI.class`".

In this example, we decided to place the header file under a "`include`" sub-directory. The generated 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

```
#include <jni.h>
#include <stdio.h>
#include "include\myjni_HelloJNI.h"

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

Compile the C program:

```
// for Windows 64-bit JDK
> x86_64-w64-mingw32-gcc -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -
shared -o hello.dll HelloJNI.c
```

You can now run the JNI program:

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

2.5 JNI in Module (JDK 9)

[TODO]

2.6 JNI in Eclipse [To Check]

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();  // Allocate an instance and 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 (Pre JDK-10)

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
# Pre JDK-10. JDK 10 removes the javah utility, need to use javac -h instead [TO
CHECK]
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` (Pre JDK-10):

```
> 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.

2.7 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-type*s:
    1. `jclass` for `java.lang.Class`.
    2. `jstring` for `java.lang.String`.
    3. `jthrowable` for `java.lang.Throwable`.
    4. `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

```
public class TestJNIPrimitive {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Declare a native method average() that receives two ints and return a
double containing the average
   private native double average(int n1, int n2);

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

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 -h . TestJNIPrimitive.java
```

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 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:

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

JNIEXPORT jdouble JNICALL Java_TestJNIPrimitive_average
          (JNIEnv *env, jobject thisObj, jint n1, jint
n2) {
   jdouble result;
   printf("In C, the numbers are %d and %d\n", n1, n2);
   result = ((jdouble)n1 + n2) / 2.0;
   // jint is mapped to int, jdouble is mapped to
double
   return result;
}
```

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

```
gcc -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o myjni.dll
TestJNIPrimitive.c
```

Now, run the Java Program:

```
java -Djava.library.path=. TestJNIPrimitive
```

C++ Implementation - TestJNIPrimitive.cpp

```
#include <jni.h>
#include <iostream>
#include "TestJNIPrimitive.h"
using namespace std;

JNIEXPORT jdouble JNICALL Java_TestJNIPrimitive_average
          (JNIEnv *env, jobject obj, jint n1, jint n2) {
   jdouble result;
   cout << "In C++, the numbers are " << n1 << " and " << n2 <<
endl;
   result = ((jdouble)n1 + n2) / 2.0;
   // jint is mapped to int, jdouble is mapped to double
   return result;
}
```

Compile the C++ program:

```
g++ -I"%JAVA_HOME%\include" -I"%JAVA_HOME%\include\win32" -shared -o myjni.dll
TestJNIPrimitive.cpp
```

4.2 Passing Strings

Java JNI Program: TestJNIString.java

```
public class TestJNIString {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }
   // Native method that receives a Java String and return a Java String
   private native String sayHello(String msg);

   public static void main(String args[]) {
      String result = new TestJNIString().sayHello("Hello from Java");
      System.out.println("In Java, the returned string is: " + result);
   }
}
```

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 -h . TestJNIString.java
```

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`.

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

JNIEXPORT jstring JNICALL Java_TestJNIString_sayHello(JNIEnv *env, jobject
thisObj, jstring inJNIStr) {
   // Step 1: Convert the JNI String (jstring) into C-String (char*)
   const char *inCStr = (*env)->GetStringUTFChars(env, inJNIStr, NULL);
   if (NULL == inCStr) return NULL;

   // Step 2: Perform its intended operations
   printf("In C, the received string is: %s\n", inCStr);
   (*env)->ReleaseStringUTFChars(env, inJNIStr, inCStr);  // release resources

   // Prompt user for a C-string
   char outCStr[128];
   printf("Enter a String: ");
   scanf("%s", outCStr);    // not more than 127 characters

   // Step 3: Convert the C-string (char*) into JNI String (jstring) and
return
   return (*env)->NewStringUTF(env, outCStr);
}
```

Compile the C program into shared library.

gcc -I"<JAVA_HOME>\include" -I"<JAVA_HOME>\include\win32" -shared -o myjni.dll TestJNIString.c

Now, run the Java Program:

```
java -Djava.library.path=. 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

```
#include <jni.h>
#include <iostream>
#include <string>
#include "TestJNIString.h"
using namespace std;

JNIEXPORT jstring JNICALL Java_TestJNIString_sayHello(JNIEnv *env, jobject
thisObj, jstring inJNIStr) {
   // Step 1: Convert the JNI String (jstring) into C-String (char*)
   const char *inCStr = env->GetStringUTFChars(inJNIStr, NULL);
   if (NULL == inCStr) return NULL;

   // Step 2: Perform its intended operations
   cout << "In C++, the received string is: " << inCStr << endl;
   env->ReleaseStringUTFChars(inJNIStr, inCStr);  // release resources

   // Prompt user for a C++ string
   string outCppStr;
   cout << "Enter a String: ";
   cin >> outCppStr;

   // Step 3: Convert the C++ string to C-string, then to JNI String (jstring)
and return
   return env->NewStringUTF(outCppStr.c_str());
}
```

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

```
g++ -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 us "`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

```
public class TestJNIPrimitiveArray {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Declare a native method sumAndAverage() that receives an int[] and
   //  return a double[2] array with [0] as sum and [1] as average
   private native double[] sumAndAverage(int[] numbers);

   // Test Driver
   public static void main(String args[]) {
      int[] numbers = {22, 33, 33};
      double[] results = new TestJNIPrimitiveArray().sumAndAverage(numbers);
      System.out.println("In Java, the sum is " + results[0]);
      System.out.println("In Java, the average is " + results[1]);
   }
}
```

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:

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

JNIEXPORT jdoubleArray JNICALL Java_TestJNIPrimitiveArray_sumAndAverage
          (JNIEnv *env, jobject thisObj, jintArray inJNIArray) {
   // Step 1: Convert the incoming JNI jintarray to C's jint[]
   jint *inCArray = (*env)->GetIntArrayElements(env, inJNIArray, NULL);
   if (NULL == inCArray) return NULL;
   jsize length = (*env)->GetArrayLength(env, inJNIArray);

   // Step 2: Perform its intended operations
   jint sum = 0;
   int i;
   for (i = 0; i < length; i++) {
      sum += inCArray[i];
   }
```

```
   jdouble average = (jdouble)sum / length;
   (*env)->ReleaseIntArrayElements(env, inJNIArray, inCArray, 0); // release
resources

   jdouble outCArray[] = {sum, average};

   // Step 3: Convert the C's Native jdouble[] to JNI jdoublearray, and return
   jdoubleArray outJNIArray = (*env)->NewDoubleArray(env, 2);  // allocate
   if (NULL == outJNIArray) return NULL;
   (*env)->SetDoubleArrayRegion(env, outJNIArray, 0 , 2, outCArray);  // copy
   return outJNIArray;
}
```

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

```
public class TestJNIInstanceVariable {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Instance variables
   private int number = 88;
   private String message = "Hello from Java";

   // Declare a native method that modifies the instance variables
   private native void modifyInstanceVariable();

   // Test Driver
   public static void main(String args[]) {
      TestJNIInstanceVariable test = new TestJNIInstanceVariable();
      test.modifyInstanceVariable();
      System.out.println("In Java, int is " + test.number);
      System.out.println("In Java, String is " + test.message);
   }
}
```

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

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

```
JNIEXPORT void JNICALL Java_TestJNIInstanceVariable_modifyInstanceVariable
          (JNIEnv *env, jobject thisObj) {
   // Get a reference to this object's class
   jclass thisClass = (*env)->GetObjectClass(env, thisObj);

   // int
   // Get the Field ID of the instance variables "number"
   jfieldID fidNumber = (*env)->GetFieldID(env, thisClass, "number", "I");
   if (NULL == fidNumber) return;

   // Get the int given the Field ID
   jint number = (*env)->GetIntField(env, thisObj, fidNumber);
   printf("In C, the int is %d\n", number);

   // Change the variable
   number = 99;
   (*env)->SetIntField(env, thisObj, fidNumber, number);

   // Get the Field ID of the instance variables "message"
   jfieldID fidMessage = (*env)->GetFieldID(env, thisClass, "message",
"Ljava/lang/String;");
   if (NULL == fidMessage) return;

   // String
   // Get the object given the Field ID
   jstring message = (*env)->GetObjectField(env, thisObj, fidMessage);

   // Create a C-string with the JNI String
   const char *cStr = (*env)->GetStringUTFChars(env, message, NULL);
   if (NULL == cStr) return;

   printf("In C, the string is %s\n", cStr);
   (*env)->ReleaseStringUTFChars(env, message, cStr);

   // Create a new C-string and assign to the JNI string
   message = (*env)->NewStringUTF(env, "Hello from C");
   if (NULL == message) return;

   // modify the instance variables
   (*env)->SetObjectField(env, thisObj, fidMessage, message);
}
```

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

```
public class TestJNIStaticVariable {
   static {
      System.loadLibrary("myjni"); // nyjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Static variables
   private static double number = 55.66;

   // Declare a native method that modifies the static variable
   private native void modifyStaticVariable();

   // Test Driver
   public static void main(String args[]) {
      TestJNIStaticVariable test = new TestJNIStaticVariable();
      test.modifyStaticVariable();
      System.out.println("In Java, the double is " + number);
   }
}
```

C Implementation - TestJNIStaticVariable.c

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

JNIEXPORT void JNICALL
Java_TestJNIStaticVariable_modifyStaticVariable
          (JNIEnv *env, jobject thisObj) {
   // Get a reference to this object's class
   jclass cls = (*env)->GetObjectClass(env, thisObj);

   // Read the int static variable and modify its value
   jfieldID fidNumber = (*env)->GetStaticFieldID(env, cls, "number",
"D");
   if (NULL == fidNumber) return;
   jdouble number = (*env)->GetStaticDoubleField(env, cls,
fidNumber);
   printf("In C, the double is %f\n", number);
   number = 77.88;
   (*env)->SetStaticDoubleField(env, cls, fidNumber, number);
}
```

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

```
public class TestJNICallBackMethod {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Declare a native method that calls back the Java methods below
   private native void nativeMethod();

   // To be called back by the native code
   private void callback() {
      System.out.println("In Java");
   }

   private void callback(String message) {
      System.out.println("In Java with " + message);
   }

   private double callbackAverage(int n1, int n2) {
      return ((double)n1 + n2) / 2.0;
   }

   // Static method to be called back
   private static String callbackStatic() {
      return "From static Java method";
   }

   // Test Driver
   public static void main(String args[]) {
      new TestJNICallBackMethod().nativeMethod();
   }
}
```

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

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

JNIEXPORT void JNICALL Java_TestJNICallBackMethod_nativeMethod
          (JNIEnv *env, jobject thisObj) {

   // Get a class reference for this object
   jclass thisClass = (*env)->GetObjectClass(env, thisObj);

   // Get the Method ID for method "callback", which takes no arg and return
void
   jmethodID midCallBack = (*env)->GetMethodID(env, thisClass, "callback", "
()V");
   if (NULL == midCallBack) return;
   printf("In C, call back Java's callback()\n");
   // Call back the method (which returns void), baed on the Method ID
   (*env)->CallVoidMethod(env, thisObj, midCallBack);

   jmethodID midCallBackStr = (*env)->GetMethodID(env, thisClass,
                              "callback", "(Ljava/lang/String;)V");
   if (NULL == midCallBackStr) return;
   printf("In C, call back Java's called(String)\n");
   jstring message = (*env)->NewStringUTF(env, "Hello from C");
   (*env)->CallVoidMethod(env, thisObj, midCallBackStr, message);

   jmethodID midCallBackAverage = (*env)->GetMethodID(env, thisClass,
                                  "callbackAverage", "(II)D");
   if (NULL == midCallBackAverage) return;
   jdouble average = (*env)->CallDoubleMethod(env, thisObj,
midCallBackAverage, 2, 3);
   printf("In C, the average is %f\n", average);

   jmethodID midCallBackStatic = (*env)->GetStaticMethodID(env, thisClass,
                                 "callbackStatic", "()Ljava/lang/String;");
   if (NULL == midCallBackStatic) return;
   jstring resultJNIStr = (*env)->CallStaticObjectMethod(env, thisClass,
midCallBackStatic);
   const char *resultCStr = (*env)->GetStringUTFChars(env, resultJNIStr,
```

```
NULL);
   if (NULL == resultCStr) return;
   printf("In C, the returned string is %s\n", resultCStr);
   (*env)->ReleaseStringUTFChars(env, resultJNIStr, resultCStr);
}
```

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 `GetStaticMethodID(), 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, 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

```
public class TestJNIConstructor {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // Native method that calls back the constructor and return the constructed
object.
   // Return an Integer object with the given int.
   private native Integer getIntegerObject(int number);

   public static void main(String args[]) {
      TestJNIConstructor obj = new TestJNIConstructor();
      System.out.println("In Java, the number is :" +
obj.getIntegerObject(9999));
   }
}
```

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

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

JNIEXPORT jobject JNICALL Java_TestJNIConstructor_getIntegerObject
          (JNIEnv *env, jobject thisObj, jint number) {
   // Get a class reference for java.lang.Integer
   jclass cls = (*env)->FindClass(env, "java/lang/Integer");

   // Get the Method ID of the constructor which takes an int
   jmethodID midInit = (*env)->GetMethodID(env, cls, "<init>", "(I)V");
   if (NULL == midInit) return NULL;
   // Call back constructor to allocate a new instance, with an int argument
   jobject newObj = (*env)->NewObject(env, cls, midInit, number);

   // Try running the toString() on this newly create object
   jmethodID midToString = (*env)->GetMethodID(env, cls, "toString", "
()Ljava/lang/String;");
   if (NULL == midToString) return NULL;
   jstring resultStr = (*env)->CallObjectMethod(env, newObj, midToString);
   const char *resultCStr = (*env)->GetStringUTFChars(env, resultStr, NULL);
```

```
      printf("In C: the number is %s\n", resultCStr);

      //May need to call releaseStringUTFChars() before return
      return newObj;
   }
```

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

```
import java.util.ArrayList;

public class TestJNIObjectArray {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }
   // Native method that receives an Integer[] and
   //  returns a Double[2] with [0] as sum and [1] as average
   private native Double[] sumAndAverage(Integer[] numbers);

   public static void main(String args[]) {
      Integer[] numbers = {11, 22, 32};  // auto-box
      Double[] results = new TestJNIObjectArray().sumAndAverage(numbers);
      System.out.println("In Java, the sum is " + results[0]);  // auto-
unbox
      System.out.println("In Java, the average is " + results[1]);
   }
}
```

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

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

JNIEXPORT jobjectArray JNICALL Java_TestJNIObjectArray_sumAndAverage
          (JNIEnv *env, jobject thisObj, jobjectArray inJNIArray) {
   // Get a class reference for java.lang.Integer
   jclass classInteger = (*env)->FindClass(env, "java/lang/Integer");
   // Use Integer.intValue() to retrieve the int
   jmethodID midIntValue = (*env)->GetMethodID(env, classInteger, "intValue",
```

```
"()I");
   if (NULL == midIntValue) return NULL;

   // Get the value of each Integer object in the array
   jsize length = (*env)->GetArrayLength(env, inJNIArray);
   jint sum = 0;
   int i;
   for (i = 0; i < length; i++) {
      jobject objInteger = (*env)->GetObjectArrayElement(env, inJNIArray, i);
      if (NULL == objInteger) return NULL;
      jint value = (*env)->CallIntMethod(env, objInteger, midIntValue);
      sum += value;
   }
   double average = (double)sum / length;
   printf("In C, the sum is %d\n", sum);
   printf("In C, the average is %f\n", average);

   // Get a class reference for java.lang.Double
   jclass classDouble = (*env)->FindClass(env, "java/lang/Double");

   // Allocate a jobjectArray of 2 java.lang.Double
   jobjectArray outJNIArray = (*env)->NewObjectArray(env, 2, classDouble,
NULL);

   // Construct 2 Double objects by calling the constructor
   jmethodID midDoubleInit = (*env)->GetMethodID(env, classDouble, "<init>", "
(D)V");
   if (NULL == midDoubleInit) return NULL;
   jobject objSum = (*env)->NewObject(env, classDouble, midDoubleInit,
(double)sum);
   jobject objAve = (*env)->NewObject(env, classDouble, midDoubleInit,
average);
   // Set to the jobjectArray
   (*env)->SetObjectArrayElement(env, outJNIArray, 0, objSum);
   (*env)->SetObjectArrayElement(env, outJNIArray, 1, objAve);

   return outJNIArray;
}
```

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

```
public class TestJNIReference {
   static {
      System.loadLibrary("myjni"); // myjni.dll (Windows) or libmyjni.so
(Unixes)
   }

   // A native method that returns a java.lang.Integer with the given int.
   private native Integer getIntegerObject(int number);

   // Another native method that also returns a java.lang.Integer with the
given int.
   private native Integer anotherGetIntegerObject(int number);

   public static void main(String args[]) {
      TestJNIReference test = new TestJNIReference();
      System.out.println(test.getIntegerObject(1));
      System.out.println(test.getIntegerObject(2));
      System.out.println(test.anotherGetIntegerObject(11));
      System.out.println(test.anotherGetIntegerObject(12));
      System.out.println(test.getIntegerObject(3));
      System.out.println(test.anotherGetIntegerObject(13));
   }
}
```

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!

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

// Global Reference to the Java class "java.lang.Integer"
static jclass classInteger;
static jmethodID midIntegerInit;

jobject getInteger(JNIEnv *env, jobject thisObj, jint number) {

   // Get a class reference for java.lang.Integer if missing
   if (NULL == classInteger) {
      printf("Find java.lang.Integer\n");
      classInteger = (*env)->FindClass(env, "java/lang/Integer");
   }
   if (NULL == classInteger) return NULL;

   // Get the Method ID of the Integer's constructor if missing
   if (NULL == midIntegerInit) {
      printf("Get Method ID for java.lang.Integer's constructor\n");
      midIntegerInit = (*env)->GetMethodID(env, classInteger, "<init>", "
(I)V");
   }
   if (NULL == midIntegerInit) return NULL;

   // Call back constructor to allocate a new instance, with an int argument
   jobject newObj = (*env)->NewObject(env, classInteger, midIntegerInit,
number);
   printf("In C, constructed java.lang.Integer with number %d\n", number);
   return newObj;
}

JNIEXPORT jobject JNICALL Java_TestJNIReference_getIntegerObject
          (JNIEnv *env, jobject thisObj, jint number) {
   return getInteger(env, thisObj, number);
}

JNIEXPORT jobject JNICALL Java_TestJNIReference_anotherGetIntegerObject
          (JNIEnv *env, jobject thisObj, jint number) {
   return getInteger(env, thisObj, number);
}
```

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. JNI Common Errors

```
ERROR MESSAGE: SEVERE: java.lang.UnsatisfiedLinkError: no xxx in java.library.path
PROBABLE CAUSES: Your program uses a native library from a 3rd-party API (such as
JOGL),
   which cannot be located in the native library search paths.
POSSIBLE SOLUTION:
A Java Native Library (JNI) contains non-Java library codes (in filetype of ".dll"
in Windows, ".so" in Linux,
   ".jnilib" in MacOS). For example, JOGL's "jogl_xxx.dll", "gluegen-rt.dll".
   These dll's are needed for proper operations.
The directory path of native libraries must be included in Java system's property
"java.library.path".
The "java.library.path" usually mirrors the Envrionment Variable PATH. You can
list the entries by issuing:
   System.out.println(System.getProperty("java.library.path"));

To include a directory in "java.library.path", you can use VM command-line option
-Djava.library.path=pathname
For JRE:
   > java -Djava.library.path=d:\bin\jogl2.0\lib myjoglapp

For Eclipse, the VM command-line option can be set in "Run Configuration..." ⇒
"Arguments" ⇒ "VM Arguments".
Alternatively, you can create a User library and specifying the native library
(Refer to "Eclipse How-To")

For NetBeans, the VM command-line option can be set in "Set Configuration" ⇒
"Customize..." ⇒ "Run" ⇒ "VM options".
```

9. 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 9.0.1, Cygwin's MinGW-w64 GCC/G++
Last modified: March, 2018

Feedback, comments, corrections, and errata can be sent to Chua Hock-Chuan (ehchua@ntu.edu.sg) | HOME