

Native Methods

The information in this section reflects the native method API in JDK 1.1 called the Java Native Interface (JNI). The API in JDK 1.0.2 is different. Microsoft uses a their own API in their Java interpreter.



Justification

- Reasons to use
 - Reuse
 - utilize existing code
 - Efficiency
 - use the language most appropriate for the task
- Reasons not to use
 - Applets
 - cannot use native methods due to security restrictions
 - Portability
 - native methods aren't portable
 - Extra work
 - must use additional tools to generate several kinds of files
 - javah to create special header files
 - compiler to create native method object files
 - various tools to create shared libraries

Using sockets to communicate with legacy applications is an alternative.



Steps To Use Native Methods

(not as straightforward as C calling FORTRAN because two different "worlds" are involved, compiled object code vs. interpreted/garbage collected Java bytecode)

- Write Java class that
 - declares native methods
 - loads a shared library of native object code
- Compile Java code
- Create header file for native methods
 - using `javah -jni class-name`
- Write native methods
 - the spec. says this can be done in "other programming languages, such as C, C++, and assembly"
- Compile native methods
- Add native object code to a UNIX shared library or Windows DLL
 - under UNIX, if the shared library is not in the current directory, it must be found in `LD_LIBRARY_PATH`
 - under Windows, if the DLL is not in the current directory, it must be found in `PATH`
- Run Java code

no examples of native methods in anything other than C and C++ were found



Java Class - HelloWorld.java

```
public class HelloWorld {  
    public native void displayHelloWorld();  
  
    static {  
        System.loadLibrary("Hello"); // reference to libHello.so or Hello.dll  
    }  
  
    public static void main(String[] args) {  
        HelloWorld hw = new HelloWorld();  
        hw.displayHelloWorld();  
    }  
}
```

extention may be different for some UNIX OSes

had to create an instance since the native method is not static

if displayHelloWorld() was a static method then creating an instance wouldn't be necessary and "HelloWorld." could be used here

- To compile this (creates HelloWorld.class)
 - javac HelloWorld.java
- To create the header file HelloWorld.h
 - javah -jni HelloWorld
 - include dot-separated package name if the HelloWorld class is in a package
 - affects name of generated header file and name of the C function



C++ Native Methods

- Must prevent name mangling
 - done in C++ to support method overloading
 - prevented with extern “C”
 - done for you in generated header file
- Can’t use member functions for native methods
- Can only use global functions



Header File - HelloWorld.h

- Generated by `javah -jni HelloWorld`

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

```
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
```

```
/*
```

```
 * Class:      HelloWorld
 * Method:     displayHelloWorld
 * Signature:  ()V
 */
```

no arguments, void return type
(signatures are explained on page 14)

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

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

the Java object on which the
native method was invoked

JNI interface pointer used
to invoke JNI functions

The JNIEXPORT and
JNICALL macros are
defined in jni_md.h
which is included
from jni.h.
It contains platform
specific definitions.

- Use this function signature in
HelloWorld.cpp

- full method name is formed from package name,
class name, and native method name



Native Method - HelloWorld.cpp

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

// The function signature can be copied from HelloWorld.h
JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv* env, jobject obj) {
    printf("Hello World!\n");
}
```

could also use `iostream.h`

includes `jni.h`

could also use
`cout << "Hello World!" << endl;`

- Compile this to create object file (HelloWorld.o)
 - under UNIX
 - `gcc -c -I$JAVAHOME/include -I$JAVAHOME/include/solaris HelloWorld.cpp`
- Add object file to shared library (libHello.so)
 - under UNIX
 - `gcc -G HelloWorld.o -o libHello.so`
 - verify that libHello.so can be found in LD_LIBRARY_PATH
- To run
 - `java HelloWorld`



Microsoft Developer Studio (Visual C++ V4)

- Steps
 - File...New...
 - choose “Project Workspace”
 - choose “Dynamic-Link Library”
 - enter a name for the project
 - specify the location where it should be created
 - click OK button
 - creates a .mdp file
 - Build...Settings...General
 - select the Debug directory and the Release directory
 - for each, make fields for “Intermediate” and “Output” files in “Output Directories” empty
 - Tools...Options...Directories
 - add x\java\include and x\java\include\win32 where x is the location where the JDK is installed
 - in the “Project Workspace” window, click on the FileView tab
 - Insert...Files into Project...
 - insert the implementation files (the C/C++ native methods)
 - Build...Build *project-name.dll*



Microsoft Developer Studio (Visual C++ V5)

- Steps
 - File...New...
 - choose “Projects” tab
 - choose “Win32 Dynamic-Link Library”
 - enter a name for the project
 - specify the location where it should be created
 - click OK button
 - creates a .mdp file
 - Project...Settings...General
 - select “Settings For: Win32 Debug” and “Settings For: Win32 Release”
 - for each, make fields for “Intermediate” and “Output” files in “Output Directories” empty
 - Tools...Options...Directories
 - add x\java\include and x\java\include\win32 where x is the location where the JDK is installed
 - in the “Project Workspace” window, click on the FileView tab
 - Right-click the “files” icon and pick “Add Files to Project ...”
 - insert the implementation files (the C/C++ native methods)
 - Build...Build *project-name.dll*



Passing Arguments To Native Methods

- Primitive types
 - passed by value
 - changes in native method aren't reflected in the Java variable
 - alternative is to pass primitives in a type wrapper object
 - converted to closest type in native language
- Reference types
 - arrays, strings, and objects
 - passed by reference
 - native types are jarray, jstring and jobject
 - these are defined in jni.h



Primitive Parameter Example

- PrimParam.java

```
public class PrimParam {
    public native double takePrimParam(int i, double d, String s);

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

    public static void main(String[] args) {
        PrimParam app = new PrimParam();
        System.out.println
            ("result is " + app.takePrimParam(19, 3.14, "Duke"));
    }
}
```

↑
not a primitive!

- PrimParam.cpp

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

JNIEXPORT jdouble JNICALL Java_PrimParam_takePrimParam
    (JNIEnv* env, jobject obj, jint i, jdouble d, jstring s) {
    cout << "i=" << i
        << ", d=" << d
        << ", s=" << env->GetStringUTFChars(s, 0)
        << endl;
    return i + d;
}
```

see p. 14 for mapping
between Java types
and native types

Unicode Text Format

↑
not interested in whether it
makes its own copy of the String



Difference Between Coding Native Methods in C vs. C++

- Comment in jni.h

```
/*  
 * We use inlined functions for C++ so that programmers  
 * can write:  
 *     env->FindClass("java/lang/String")  
 * in C++ rather than:  
 *     (*env)->FindClass(env, "java/lang/String")  
 * in C.  
 */
```

"The Java Virtual Machine Specification" p. 89 says
"For historical reasons ... the periods that normally
separate the identifiers that make up the fully
qualified name are replaced by forward slashes."

- When invoking JNI functions from C use

```
(*env)->function(env, ...);
```

- When invoking JNI functions from C++
use

```
env->function(...);
```

- Examples in these notes use the C++ form



Object Parameter Example

- ObjParam.java

```
import java.util.Vector;

public class ObjParam {
    // passing a Vector containing Double objects
    public native double sum(Vector v);

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

    public static void main(String[] args) {
        ObjParam app = new ObjParam();

        Vector v = new Vector();
        v.addElement(new Double(3.14));
        v.addElement(new Double(19.2));
        v.addElement(new Double(6.7));

        System.out.println("result is " + app.sum(v));
    }
}
```



Object Parameter Example (Cont'd)

- **ObjParam.cpp**

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

a **jmethodId** variable naming convention - concatenate the name of the Java class and the name of the method

```
JNIEXPORT jdouble JNICALL Java_ObjParam_sum
    (JNIEnv *env, jobject obj, jobject vector) {

    jclass vectorClass = env->FindClass("java/util/Vector");
    jclass doubleClass = env->FindClass("java/lang/Double");

    jmethodID vectorSize = env->GetMethodID(vectorClass, "size", "()I");

    int size = env->CallIntMethod(vector, vectorSize);
    cout << "size of Vector is " << size << endl;

    jmethodID vectorElementAt =
        env->GetMethodID(vectorClass, "elementAt", "(I)Ljava/lang/Object;");
    jmethodID doubleDoubleValue =
        env->GetMethodID(doubleClass, "doubleValue", "()D");

    jdouble sum = 0;
    jobject object;
    jdouble value;
    for (int i = 0; i < size; ++i) {
        cout << "calling elementAt(" << i << ")" << endl;
        object = env->CallObjectMethod(vector, vectorElementAt, i);
        value = env->CallDoubleMethod(object, doubleDoubleValue);
        cout << "value = " << value << endl;

        sum += value;
    }

    return sum;
}
```

alternative is
env->GetObjectClass
(vector);

signatures are explained on the next page

gets the double primitive value from a Double type wrapper object

could maintain data persistence on the C/C++ side by allocating space for data and keeping a pointer to it in a static variable



Java Method Signature Strings

Java Type	Native Type	Type Signature*
boolean	jboolean	Z
byte	jbyte	B
char	jchar	C
short	jshort	S
int	jint	I
long	jlong	J
float	jfloat	F
double	jdouble	D
void	void	V
type[]	jarray	[type
fully-qualified-class	jobject	Lfully-qualified-class;
String	jstring	Ljava/lang/String;

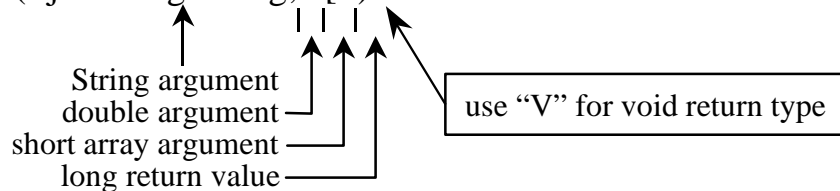
* Type signatures originate from FieldDescriptors in Java bytecode files.
See "The Java Virtual Machine Specification" p. 90-91.

- Method signatures are required for the last argument to GetMethodId

– example

- if the method is
 - long callMe(String s, double d, short[] a)
- the method signature is

– “(Ljava/lang/String;D[S)J”



When invoking JNI functions from C use (*env)->function(env, ...).
When invoking JNI functions from C++ use env->function(...).

JNI Functions For Classes and Objects

(not a full list)

- FindClass

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

- GetObjectClass

```
jclass GetObjectClass(JNIEnv* env, jobject obj)
```

- NewObject

```
jobject NewObject(JNIEnv* env, jclass clazz,  
                  jmethodID methodID, ...)
```

- `clazz` is the class of the object to be created
- `methodID` is the constructor to be used
 - get with `GetMethodID` (see page 18)
 - replace ... with the constructor arguments

- IsInstanceOf

```
jboolean IsInstanceOf(JNIEnv* env, jobject obj, jclass clazz)
```



JNI Functions For Object Fields

(not a full list)

- **GetFieldID**

```
jfieldID GetFieldID(JNIEnv* env, jclass clazz,  
                    const char* fieldName,  
                    const char* signature)
```

- **GetField**

```
NativeType Get?Field(JNIEnv* env, jobject obj,  
                     jfieldID fieldID)
```

type of the field

see p. 14

– replace ? with Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double

- **SetField**

```
void Set?Field(JNIEnv* env, jobject obj,  
               jfieldID fieldID, NativeType value)
```

– replace ? with Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double

- **Can get and set all fields of the class of the native method regardless of access specifiers**

– that's correct because native methods are methods of the class that contains the fields



JNI Functions For Class Fields

(not a full list)

- **GetStaticFieldID**

```
jfieldID GetStaticFieldID(JNIEnv* env, jclass clazz,  
                          const char* fieldName,  
                          const char* signature)
```

- **GetStaticField**

```
NativeType GetStatic?Field(JNIEnv* env, jclass clazz,  
                           jfieldID fieldID)
```

- replace ? with Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double

- **SetStaticField**

```
void SetStatic?Field(JNIEnv* env, jclass clazz,  
                    jfieldID fieldID, NativeType value)
```

- replace ? with Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double



JNI Functions For Object Methods

(not a full list)

- **GetMethodID**

```
jmethodID GetMethodID(JNIEnv* env, jclass clazz,  
                        const char* name,  
                        const char* signature)
```

- for constructors use “<init>” for name and “V” for return type in signature

- **CallMethod**

```
NativeType Call?Method(JNIEnv* env, jobject obj,  
                        jmethodID methodID, ...)
```

- replace ? with Void, Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double
- replace ... with the method arguments
- use `NewObject` to invoke constructors



JNI Functions For Class Methods

(not a full list)

- **GetStaticMethodId**

```
jmethodID GetStaticMethodID(JNIEnv* env, jclass clazz,  
                             const char* name,  
                             const char* signature)
```

- **CallStaticMethod**

```
NativeType CallStaticMethod(JNIEnv* env, jclass clazz,  
                             jmethodID methodID, ...)
```

- replace ? with Void, Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double
- replace ... with the method arguments



These are the functions that work with single-byte characters in the native method. They convert between UTF (used by Java) and ASCII (used by native methods). There are also non-UTF versions that support two-byte Unicode characters.

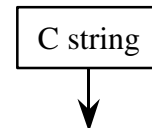
JNI Functions For Strings

(not a full list)

- **NewStringUTF**

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

- to create a Java String from a C string
- useful to pass to a Java method



- **GetStringUTFLength**

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

- to get the length of a Java String

- **GetStringUTFChars**

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

- to create a C string from a Java String
- isCopy is set to indicate whether a copy was made
 - VM dependent, you don't decide
- can cast away const to modify the C string
 - changes are not reflected in Java
 - what happens if isCopy is false and you modify the C string?



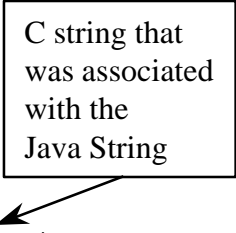
JNI Functions For Strings (Cont'd)

- **ReleaseStringUTFChars**

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

- decrements the reference count of the Java String
 - supports garbage collection
- frees the memory allocated for the C string
- doesn't change the Java String to match the C string
 - Java Strings are immutable!

C string that
was associated
with the
Java String



JNI Functions For Arrays

(not a full list)

jarray must be jbooleanArray, jbyteArray, jcharArray, jshortArray, jintArray, jlongArray, jfloatArray, jdoubleArray, or jobjectArray

- **GetArrayLength**

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

- **Scalar here means "not an object"**

- **NewScalarArray**

```
jarray New?Array(JNIEnv* env, jsize length)
```

- replace ? with Boolean, Byte, Char, Short, Int, Long, Float, or Double
- useful for constructing a Java array needed for a method argument
- can't populate with standard C/C++ array syntax
 - it's a jobject underneath
 - to populate it, create a native array by calling `GetScalarArrayElements()` and populate the native array with standard C/C++ array syntax (see next page)
 - when finished populating, release it by calling `ReleaseScalarArrayElements()`



JNI Functions For Arrays (Cont'd)

(not a full list)

- **GetScalarArrayElements**

NativeArrayType Get?ArrayElements

(JNIEnv* env, jarray array, jboolean* isCopy)

- replace ? with Boolean, Byte, Char, Short, Int, Long, Float, or Double
- allows a Java array of scalar elements to be accessed as a native array
- isCopy is set to indicate whether a copy was made
 - VM dependent, you don't decide
- call the corresponding ReleaseScalarArrayElements function when finished



JNI Functions For Arrays (Cont'd)

(not a full list)

- **ReleaseScalarArrayElements**

```
void Release?ArrayElements  
    (JNIEnv* env, jarray array,  
     NativeArrayType elems, jint mode)
```

- replace ? with Boolean, Byte, Char, Short, Int, Long, Float, or Double
- supports garbage collection of Java arrays
 - call regardless of whether a copy was made
- set mode to
 - 0 to copy the native array back to the Java array and free the space for the native array
 - JNI_COMMIT to copy back and not free
 - JNI_ABORT to not copy back and free
- must be called if a copy was made to copy array values back to the Java array
 - see isCopy argument to GetScalarArrayElements
 - test whether isCopy == JNI_TRUE



JNI Functions For Object Arrays

- **NewObjectArray**

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

- useful for constructing a Java array needed for a method argument
- can't populate with standard C/C++ array syntax
 - use SetObjectArrayElement below

- **GetObjectArrayElement**

```
jobject GetObjectArrayElement  
(JNIEnv* env, jarray array, jsize index)
```

- **SetObjectArrayElement**

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



JNI Functions For Exceptions

(not a full list)

- **ThrowNew**

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

- throws a new exception object
- `clazz` is the class of the exception

- **Throw**

```
jint Throw(JNIEnv* env, jthrowable obj)
```

- throws an existing exception object
- necessary to throw exception objects where more than a message must be passed to the constructor

- **More on these methods**

- returns control to the Java interpreter where the exception can be caught
ONLY WHEN THE NATIVE METHOD EXITS!
 - allows native code to cleanup before returning
- returns zero if successful
 - spec. doesn't say how this could fail but it could fail if
 - `clazz` doesn't represent an exception class
 - the native method isn't declared to throw the type of exception being thrown



JNI Functions For Exceptions (Cont'd)

- Calling another JNI method after throwing an exception
 - clears the exception
 - may throw a different exception
- Other useful exception-related methods

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

- returns NULL if no exception has occurred
- use `IsInstanceOf` function to determine what type of exception was thrown

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

- call if exception has been handled in native code

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

- prints exception message and stack trace



JNI Functions For Synchronization of Threads

- **MonitorEnter**

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

- synchronizes on obj
- same as entering a synchronized block in Java
- blocks until the lock is obtained
- returns zero if successful in obtaining the lock
 - spec. doesn't say how this could stop waiting and fail

- **MonitorExit**

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

- same as exiting from a synchronized block in Java
- returns zero if successful in releasing the lock
 - spec. doesn't say how this could fail but it could fail if there is not currently a lock on the object

