CHAPTER 5

JNI Technology

The Java™ platform is relatively new, which means there could be times when you will need to integrate programs written in the Java programming language with existing non-Java language services, API toolkits, and programs. The Java platform provides the Java Native Interface (JNI) to help ease this type of integration.

The JNI defines a standard naming and calling convention so the Java virtual machine can locate and invoke native methods. In fact, JNI is built into the Java virtual machine so the Java virtual machine can invoke local system calls to perform input and output, graphics, networking, and threading operations on the host operating system.

This chapter explains how to use JNI in programs written in the Java programming language to call any libraries on the local machine, call Java methods from inside native code, and explains how to create and run a Java virtual machine instance. To show how you can put JNI to use, the examples in this chapter include integrating JNI with the Xbase C++ database API, and how you can call a mathematical function. Xbase (http://www.startech.keller.tx.us/xbase/xbase.html) has sources you can download.

Covered in this Chapter

JNI Example

The ReadFile example program shows how you can use the Java Native Interface (JNI) to invoke a native method that makes C function calls to map a file into memory.

About the Example

You can call code written in any programming language from a program written in the Java programming language by declaring a native Java method, loading the library that contains the native code, and calling the native method. The ReadFile source code below does exactly that.

However, successfully running the program requires a few additional steps beyond compiling the Java programming language source file. After you compile, but before you run the example, you have to generate a header file. The native code implements the function definitions contained in the generated header file and implements the business logic as well. The following sections walk through all the steps.

```
import java.util.*;

class ReadFile {
//Native method declaration
  native byte[] loadFile(String name);
//Load the library
  static {
    System.loadLibrary("nativelib");
  }

  public static void main(String args[]) {
    byte buf[];
//Create class instance
    ReadFile mappedFile=new ReadFile();
//Call native method to load ReadFile.java
    buf=mappedFile.loadFile("ReadFile.java");
//Print contents of ReadFile.java
    for(int i=0;i<buf.length;i++) {
      System.out.print((char)buf[i]);
    }
  }
}
```

Native Method Declaration

The native declaration provides the bridge to run the native function in the Java virtual machine. In this example, the loadFile function maps to a C function called Java_ReadFile_loadFile. The function implementation accepts a String that represents a file name and returns the contents of that file in the byte array.

```
native byte[] loadFile(String name);
```

Load the Library

The library containing the native code implementation is loaded by a call to System.loadLibrary(). Placing this call in a static initializer ensures this library is only loaded once per class. The library can be loaded outside of the static block if your application requires it.

Note: You might need to configure your environment so the loadLibrary method can find your native code library. See Compile the Dynamic or Shared Object Library (page 211) for this information.

```
//API Ref      :static void loadLibrary(String libraryname)
        static {
          System.loadLibrary("nativelib");
        }
```

Compile the Program

To compile the program, run the javac compiler command as you normally would:

```
javac ReadFile.java
```

Next, you need to generate a header file with the native method declaration and implement the native method to call the C functions for loading and reading a file.

Generate the Header File

To generate a header file, run the javah command on the ReadFile class. In this example, the generated header file is named ReadFile.h. It provides a method signature that you have to use when you implement the loadfile native function.

```
javah -jni ReadFile
```

Method Signature

The ReadFile.h header file defines the interface to map the Java method to the native C function. It uses a method signature to map the arguments and return value of the Java mappedfile.loadFile method to the loadFile native method in the nativelib library. Here is the loadFile native method mapping (method signature):

```
/*
 * Class:     ReadFile
 * Method:    loadFile
 * Signature: (Ljava/lang/String;)[B
 */
JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
  (JNIEnv *, jobject, jstring);
```

The method signature parameters function as follows:

- JNIEnv *: A pointer to the JNI environment. This pointer is a handle to the current thread in the Java virtual machine, and contains mapping and other housekeeping information.
- jobject: A reference to the object that called this native code. If the calling method is static, this parameter would be type jclass instead of jobject.
- jstring: The parameter supplied to the native method. In this example, it is the name of the file to be read.

Implement the Native Method

In this native C source file, the loadFile definition is a copy and paste of the C declaration contained in ReadFile.h. The definition is followed by the native method implementation. JNI provides a mapping for both C and C++ by default.

```
      /* file nativelib.c */
      #include <jni.h>
      #include <sys/types.h>
      #include <sys/ipc.h>
      #include <sys/shm.h>
      #include <sys/mman.h>
      #include <sys/stat.h>
      #include <fcntl.h>
      #include <unistd.h>

      JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
        (JNIEnv * env, jobject jobj, jstring name) {
          caddr_t m;
          jbyteArray jb;
          jboolean iscopy;
          struct stat finfo;
//API Ref     :const char* GetStringUTFChars(JNIEnv *env, jstring string, jboolean *iscopy)
          const char *mfile = (*env)->GetStringUTFChars(env, name, &iscopy);
          int fd = open(mfile, O_RDONLY);
          if (fd == -1) {
            printf("Could not open %s\n", mfile);
          }
          lstat(mfile, &finfo);
          m = mmap((caddr_t) 0, finfo.st_size,
                      PROT_READ, MAP_PRIVATE, fd, 0);
            if (m == (caddr_t)-1) {
              printf("Could not mmap %s\n", mfile);
              return(0);
            }
//API Ref      :jbyteArray NewByteArray(JNIEnv *env, jsize length)
            jb=(*env)->NewByteArray(env, finfo.st_size);
//API Ref      :SetByteArrayRegion(JNIEnv *env, jbyteArray array, jsize startelement, jsize length, jbyte *buffer)
           (*env)->SetByteArrayRegion(env, jb, 0, finfo.st_size, (jbyte *)m);
            close(fd);
            (*env)->ReleaseStringUTFChars(env, name, mfile);
            return (jb);
        }
```

You can approach calling an existing C function instead of implementing one in one of two ways:

1. Map the name generated by JNI to the existing C function name. The Other Programming Issues (page 220) section shows how to map between Xbase database functions and Java programming language code
2. Use the shared stubs code available from the JNI page (http://java.sun.com/products/jdk/faq/jni-faq.html) on the java.sun.com web site.

Compile the Dynamic or Shared Object Library

The library needs to be compiled as a dynamic or shared object library so it can be loaded at runtime. Static or archive libraries are compiled into an executable and cannot be loaded at runtime. The shared object or dynamic library for the loadFile example is compiled on various platforms as follows:

GNU C/Linux

```
gcc  -o libnativelib.so -shared -Wl,-soname,libnative.so  -I/export/home/jdk1.2/
include -I/export/home/jdk1.2/include/linux nativelib.c -static -lc
```

SunPro C/Solaris

```
cc -G -so libnativelib.so -I/export/home/jdk1.2/include -I/export/home/jdk1.2/include/solaris nativelib.c
```

GNU C++/Linux with Xbase

```
g++ -o libdbmaplib.so -shared -Wl,-soname,libdbmap.so  -I/export/home/jdk1.2/include -I/export/home/jdk1.2/include/
linux dbmaplib.cc -static -lc -lxbase
```

Win32/WinNT/Win2000

```
cl -Ic:/jdk1.2/include -Ic:/jdk1.2/include/win32 -LD nativelib.c -Felibnative.dll
```

Run the Example

To run the example, the Java virtual machine needs to find the native JNI library that was created. To do this, set the library path to the current directory as follows:

Unix or Linux:

```
LD_LIBRARY_PATH=.
export LD_LIBRARY_PATH
```

Windows NT/2000/95:

```
set PATH=%path%;.
```

With the library path properly specified for your platform, invoke the program as you normally would with the interpreter command:

```
java ReadFile
```

Strings and Arrays

This section explains how to pass string and array data between a program written in the Java programming language and a program written in another languages.

Passing Strings

The String object in the Java programming language, which is represented as jstring in Java Native Interface (JNI), is a 16 bit unicode string. In C a string is by default constructed from 8 bit characters. So, to access a Java String object passed to a C or C++ function or return a C or C++ string to a Java method, you need to use JNI conversion functions in your native method implementation.

The GetStringUTFChar function retrieves 8-bit characters from a 16-bit jstring using the Unicode Transformation Format (UTF). UTF represents Unicode as a string of 8 or 16 bit characters without losing any information. The third parameter GetStringUTFChar returns the result JNI_TRUE if it made a local copy of the jstring or JNI_FALSE otherwise.

C Version:

```
(*env)->GetStringUTFChars(env, name, iscopy)
```

C++ Version:

```
env->GetStringUTFChars(name, iscopy)
```

The following C JNI function converts an array of C characters to a jstring:

```
//API Ref      :jstring NewStringUTF(JNIEnv *env, const char *bytes)
    (*env)->NewStringUTF(env, lastfile)
```

The example below converts the lastfile[80] C character array to a jstring, which is returned to the calling Java method:

```
    static char lastfile[80];

    JNIEXPORT jstring JNICALL Java_ReadFile_lastFile
      (JNIEnv *env, jobject jobj) {
//API Ref      :jstring NewStringUTF(JNIEnv *env, const char *bytes)
       return((*env)->NewStringUTF(env, lastfile));
    }
```

To let the Java virtual machine know you are finished with the UTF representation, call the ReleaseStringUTFChars conversion function as shown below. The second argument is the original jstring value used to construct the UTF representation, and the third argument is the reference to the local representation of that String.

```
(*env)->ReleaseStringUTFChars(env, name, mfile);
```

If your native code can work with Unicode, without needing the intermediate UTF representation, call the GetStringChars function to retrieve the unicode string, and release the reference with a call to ReleaseStringChars:

```
    JNIEXPORT jbyteArray JNICALL Java_ReadFile_loadFile
      (JNIEnv * env, jobject jobj, jstring name) {
        caddr_t m;
        jbyteArray jb;
        struct stat finfo;
        jboolean iscopy;
//API Ref      :const jchar *GetStringChars(JNIEnv *env,  jstring string, jboolean *iscopy)
        const jchar *mfile = (*env)->GetStringChars(env, name, &iscopy);
      //...
//API Ref      :void ReleaseStringChars(JNIEnv *env, jstring string, const jchar *chars)
        (*env)->ReleaseStringChars(env, name, mfile);
```

Passing Arrays

In the example presented in the last section, the loadFile native method returns the contents of a file in a byte array, which is a primitive type in the Java programming language. You can retrieve and create primitive types in the Java programming language by calling the appropriate TypeArray function.

For example, to create a new array of floats, call NewFloatArray, or to create a new array of bytes, call NewByteArray. This naming scheme extends to retrieving elements from, adding elements to, and changing elements in the array. To get elements from an array of bytes, call GetByteArrayElements. To add elements to or change elements in the array, call Set<type>ArrayElements.

The GetByteArrayElements function affects the entire array. To work on a portion of the array, call GetByteArrayRegion instead. There is only a Set<type>ArrayRegion function for changing array elements. However the region could be of size array.length, which is equivalent to the non-existent Sete<type>ArrayElements.

Native Code Type	**Functions Used**
jboolean	NewBooleanArray GetBooleanArrayElements GetBooleanArrayRegion/SetBooleanArrayRegion ReleaseBooleanArrayRegion
jbyte	NewByteArray GetByteArrayElements GetByteArrayRegion/SetByteArrayRegion ReleaseByteArrayRegion
jchar	NewCharArray GetCharArrayElements GetCharArrayRegion/SetCharArrayRegion ReleaseCharArrayRegion
jdouble	NewDoubleArray GetDoubleArrayElements GetDoubleArrayRegion/SetDoubleArrayRegion ReleaseDoubleArrayRegion
jfloat	NewFloatArray GetFloatArrayElements GetFloatArrayRegion/SetFloatArrayRegion ReleaseFloatArrayRegion
jint	NewIntArray GetIntArrayElements GetIntArrayRegion/SetIntArrayRegion ReleaseIntArrayRegion

Native Code Type	Functions Used
jlong	NewLongArray GetLongArrayElements GetLongArrayRegion/SetLongArrayRegion ReleaseLongArrayRegion
jobject	NewObjectArray GetObjectArrayElement/SetObjectArrayElement
jshort	NewShortArray GetShortArrayElements GetShortArrayRegion/SetShortArrayRegion ReleaseShortArrayRegion

In the loadFile native method from the example in the previous section, the entire array is updated by specifying a region that is the size of the file being read in:

```
    jbyteArray jb;
//API Ref    :jbyteArray NewByteArray(JNIEnv *env, jsize length)
    jb=(*env)->NewByteArray(env, finfo.st_size);
//API Ref    :SetByteArrayRegion(JNIEnv *env, jbyteArray array, jsize startelement, jsize length, jbyte *buffer)
    (*env)->SetByteArrayRegion(env, jb, 0, finfo.st_size, (jbyte *)m);
    close(fd);
```

The array is returned to the calling Java programming language method, which in turn, garbage collects the reference to the array when it is no longer used. The array can be explicitly freed with the following call.

```
//API Ref    :void ReleaseByteArrayElements(JNIEnv *env, jbyteArray array, jbyte *elems, jint mode)
    (*env)-> ReleaseByteArrayElements(env, jb, (jbyte *)m, 0);
```

The last argument to the ReleaseByteArrayElements function above can have the following values:

- 0: Updates to the array from within the C code are reflected in the Java programming language copy.
- JNI_COMMIT: The Java programming language copy is updated, but the local jbyteArray is not freed.
- JNI_ABORT: Changes are not copied back, but the jbyteArray is freed. The value is used only if the array is obtained with a get mode of JNI_TRUE meaning the array is a copy.

See Memory Issues (page 227) for more information on memory management.

Pinning Array

When retrieving an array, you can specify if this is a copy (JNI_TRUE) or a reference to the array residing in your Java program (JNI_FALSE). If you use a reference to the array, you will want the array to stay where it is in the Java heap and not get moved by the garbage collector when it compacts heap memory. To prevent the array references from being moved, the Java virtual machine pins the array into memory. Pinning the array ensures that when the array is released, the correct elements are updated in the Java virtual machine.

In the loadfile native method example from the previous section, the array is not explicitly released. One way to ensure the array is garbage collected when it is no longer needed is to call a Java method, pass the byte array instead, and free the local array copy. This technique is shown in the section on MultiDimensional Arrays (page 217).

Object Arrays

You can store any Java object in an array with the NewObjectArray and SetObjectArrayElement function calls. The main difference between an object array and an array of primitive types is that when constructing a jobjectArray, the Java class is used as a parameter.

This next C++ example shows how to call NewObjectArray to create an array of String objects. The size of the array is set to five, the class definition is returned from a call to FindClass, and the elements of the array are initialized with an empty string. The elements of the array are updated by calling SetObjectArrayElement with the position and value to put in the array.

```
    /*  ReturnArray.C  */
    #include <jni.h>
    #include "ArrayHandler.h"

     JNIEXPORT jobjectArray JNICALL Java_ArrayHandler_returnArray
                                  (JNIEnv *env, jobject jobj){
      jobjectArray ret;
      int i;
      char *message[5]= {"first", "second", "third", "fourth", "fifth"};
      ret = (jobjectArray)env->NewObjectArray(5,
//API Ref     :jclass FindClass(JNIenv *env, const char *name)
          env->FindClass("java/lang/String"),
//API Ref     :jstring NewStringUTF(JNIEnv *env, const char *bytes)
          env->NewStringUTF(""));
      for(i=0;i<5;i++) {
//API Ref     :void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)
         env->SetObjectArrayElement(ret,i,env->NewStringUTF(message[i]));
      }
      return(ret);
     }
```

The Java class that calls this native method is as follows:

```
        // ArrayHandler.java
        public class ArrayHandler {
          public native String[] returnArray();
          static{
//API Ref    :static void loadLibrary(String libraryname)
            System.loadLibrary("nativelib");
          }
          public static void main(String args[]) {
            String ar[];
            ArrayHandler ah= new ArrayHandler();
            ar = ah.returnArray();
            for(int i=0; i<5; i++) {
              System.out.println("array element"+i+ "=" + ar[i]);
            }
          }
         }
```

MultiDimensional Arrays

You might need to call existing numerical and mathematical libraries such as the linear algebra library CLAPACK/LAPACK or other matrix crunching programs from your Java program using native methods. Many of these libraries and programs use two-dimensional and higher order arrays.

In the Java programming language, any array that has more than one dimension is treated as an array of arrays. For example, a two-dimensional integer array is handled as an array of integer arrays. The array is read horizontally, or in what is also termed as row order.

Other languages such as FORTRAN use column ordering so extra care is needed if your program hands a Java array to a FORTRAN function. Also, the array elements in an application written in the Java programming language are not guaranteed to be contiguous in memory. Some numerical libraries use the knowledge that the array elements are stored next to each other in memory to perform speed optimizations, so you might need to make an additional local copy of the array to pass to those functions.

The next example passes a two-dimensional array to a native method which then extracts the elements, performs a calculation, and calls a Java method to return the results. The array is passed as an object array that contains an array of jints. The individual elements are extracted by first retrieving a jintArray instance from the object array by calling GetObjectArrayElement, and then extracting the elements from the jintArray row.

The example uses a fixed-size matrix. If you do not know the size of the array being used, the GetArrayLength(array) function returns the size of the outermost array. You will need to call the GetArrayLength(array) function on each dimension of the array to discover the total size of the array. The new array sent back to the program written in the Java language is built in reverse. First, a jintArray instance is created and that instance is set in the object array by calling SetObjectArrayElement.

```
public class ArrayManipulation {
  private int arrayResults[][];
  Boolean lock=new Boolean(true);
  int arraySize=-1;

  public native void manipulateArray(int[][] multiplier, Boolean lock);
  static{
//API Ref    :static void loadLibrary(String libraryname)
    System.loadLibrary("nativelib");
  }
  public void sendArrayResults(int results[][]) {
    arraySize=results.length;
    arrayResults=new int[results.length][];
    System.arraycopy(results,0,arrayResults, 0, arraySize);
  }
  public void displayArray() {
    for (int i=0; i<arraySize; i++) {
      for(int j=0; j <arrayResults[i].length;j++) {
        System.out.println("array element "+i+","+j+ "= " + arrayResults[i][j]);
      }
    }
  }
  public static void main(String args[]) {
    int[][] ar = new int[3][3];
    int count=3;
    for(int i=0;i<3;i++) {
      for(int j=0;j<3;j++) {
        ar[i][j]=count;
      }
      count++;
    }
    ArrayManipulation am= new ArrayManipulation();
    am.manipulateArray(ar, am.lock);
    am.displayArray();
  }
}

#include <jni.h>
#include <iostream.h>
#include "ArrayManipulation.h"

JNIEXPORT void
     JNICALL Java_ArrayManipulation_manipulateArray
          (JNIEnv *env, jobject jobj, jobjectArray elements, jobject lock){
  jobjectArray ret;
  int i,j;
  jint arraysize;
  int asize;
  jclass cls;
  jmethodID mid;
```

```
          jfieldID fid;
          long localArrayCopy[3][3];
          long localMatrix[3]={4,4,4};

          for(i=0; i<3; i++) {
//API Ref      :jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)
            jintArray oneDim= (jintArray)env->GetObjectArrayElement(elements, i);
//API Ref      :jint* GetIntArrayElements(JNIEnv *env, jintArray array, jboolean *iscopy)
            jint *element=env->GetIntArrayElements(oneDim, 0);
            for(j=0; j<3; j++) {
              localArrayCopy[i][j]= element[j];
            }
          }
        // With the C++ copy of the array,
        // process the array with LAPACK, BLAS, etc.

          for (i=0;i<3;i++) {
            for (j=0; j<3 ; j++) {
             localArrayCopy[i][j]=
              localArrayCopy[i][j]*localMatrix[i];
            }
          }
        // Create array to send back
//API Ref      :jintArray NewIntArray(JNIEnv *env, jsize length)
          jintArray row= (jintArray)env->NewIntArray(3);
//API Ref      :jclass GetObjectClass(JNIEnv *env, jobject obj)
          ret=(jobjectArray)env->NewObjectArray(3, env->GetObjectClass(row), 0);

          for(i=0;i<3;i++) {
            row= (jintArray)env->NewIntArray(3);
//API Ref      :SetIntArrayRegion(JNIEnv *env, jintArray array, jsize startelement, jsize length, jint *buffer)
            env->SetIntArrayRegion((jintArray)row,(jsize)0,3,(jint *)localArrayCopy[i]);
            env->SetObjectArrayElement(ret,i,row);
          }
          cls=env->GetObjectClass(jobj);
//API Ref      :jmethodID GetMethodId(JNIEnv *env, jclass class, const char *methodname, const char *methodsig)
          mid=env->GetMethodID(cls, "sendArrayResults",
                              "([[I)V");
          if (mid == 0) {
            cout <<"Can't find method sendArrayResults";
            return;
          }
//API Ref      :void ExceptionClear(JNIEnv *env)
          env->ExceptionClear();
//API Ref      :jint MonitorEnter(JNIEnv *env, jobject object)
          env->MonitorEnter(lock);
//API Ref      :CallVoidMethod(JNIEnv *env, jobject object, jmethodId methodid, object arg1)
          env->CallVoidMethod(jobj, mid, ret);
//API Ref      :jint MonitorExit(JNIEnv *env, jobject object)
          env->MonitorExit(lock);
//API Ref      :jthrowable ExceptionOccurred(JNIEnv *env)
          if(env->ExceptionOccurred()) {
            cout << "error occured copying array back" << endl;
//API Ref      :void ExceptionDescribe(JNIEnv *env)
          env->ExceptionDescribe();
          env->ExceptionClear();
        }
//API Ref      :jfieldID GetFieldID(JNIEnv *env, jclass class, const char *fieldname, const char *fieldsig)
        fid=env->GetFieldID(cls, "arraySize",  "I");
        if (fid == 0) {
          cout <<"Can't find field arraySize";
          return;
        }
        asize=env->GetIntField(jobj,fid);
        if(!env->ExceptionOccurred()) {
          cout<< "Java array size=" << asize << endl;
        } else {
          env->ExceptionClear();
        }
        return;
      }
```

Other Programming Issues

This section presents information on accessing classes, methods, and fields, and covers threading, memory, and Java virtual machine issues.

Language issues

So far, the native method examples have covered calling standalone C and C++ functions that either return a result or modify parameters passed into the function. However, C++ like the Java programming language, uses instances of classes. If you create a class in one native method, the reference to this class does not have an equivalent class in the Java programming language. This makes it difficult to call functions on the C++ class that was first created.

One way to handle this situation is to keep a record of the C++ class reference and pass that back to a proxy or to the calling program. To ensure the C++ class persists across native method calls, use the C++ new operator to create a reference to the C++ object on the heap.

The following code provides a mapping between the Xbase database and Java code. The Xbase database has a C++ API and uses an initialization class to perform subsequent database operations. When the class object is created, a pointer to this object is returned as a Java int value. You can use a long or larger value for machines with greater than 32 bits.

```
public class CallDB {
  public native int initdb();
  public native short opendb(String name, int ptr);
  public native short GetFieldNo(String fieldname, int ptr);

  static {
```

```
//API Ref      :static void loadLibrary(String libraryname)
      System.loadLibrary("dbmaplib");
    }
    public static void main(String args[]) {
      String prefix=null;
      CallDB db=new CallDB();
      int res=db.initdb();
      if(args.length>=1) {
        prefix=args[0];
      }
      System.out.println(db.opendb("MYFILE.DBF", res));
      System.out.println(db.GetFieldNo("LASTNAME", res));
      System.out.println(db.GetFieldNo("FIRSTNAME", res));
     }
  }
```

The return result from the call to the initdb native method, the int value, is passed to subsequent native method calls. The native code included in the dbmaplib.cc library dereferences the Java object passed in as a parameter and retrieves the object pointer. The line xbDbf* Myfile=(xbDbf*)ptr; casts the int pointer value to be a pointer of Xbase type xbDbf.

```
#include <jni.h>
#include <xbase/xbase.h>
#include "CallDB.h"

JNIEXPORT jint JNICALL Java_CallDB_initdb(JNIEnv *env, jobject jobj) {
  xbXBase* x;
  x= new xbXBase();
  xbDbf* Myfile;
  Myfile =new xbDbf(x);
  return ((jint)Myfile);
}
JNIEXPORT jshort JNICALL Java_CallDB_opendb(JNIEnv *env, jobject jobj,
                                 jstring dbname, jint ptr) {
  xbDbf* Myfile=(xbDbf*)ptr;
  return((*Myfile).OpenDatabase( "MYFILE.DBF"));
}
JNIEXPORT jshort JNICALL Java_CallDB_GetFieldNo(JNIEnv *env, jobject jobj,
                                   jstring fieldname, jint ptr) {
  xbDbf* Myfile=(xbDbf*)ptr;
  return((*Myfile).GetFieldNo(env->GetStringUTFChars(fieldname,0)));
}
```

Calling Methods

The section on arrays highlighted some reasons for calling Java programming language methods from within native code; for example, when you need to free the result you intend to return. Other uses for calling Java native methods from within your native code are if you need to return more than one result or you just simply want to modify Java programming language values from within native code. Calling a Java programming language method from within native code involves the following three steps:

1. Retrieve a class reference
2. Retrieve a method identifier
3. Call the Methods

Retrieve a Class Reference

The first step is to retrieve a reference to the class that contains the methods you want to access. To retrieve a reference, you can either use the FindClass method or access the jobject or jclass argument to the native method.

Use the FindClass method:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jobject jobj){
  jclass cls = (*env)->FindClass(env, "ClassName");
}
```

Use the jobject argument:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jobject jobj){
  jclass cls=(*env)->GetObjectClass(env, jobj);
}
```

or
Use the jclass argument:

```
JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jclass jcls){
  jclass cls=jcls;
}
```

Retrieve a Method Identifier

Once the class has been obtained, the second step is to call the GetMethodID function to retrieve an identifier for a method you select in the class. The identifier is needed when calling the method of that class instance. Because the Java programming language supports method overloading, you also need to specify the method signature you want to call. To find out what signature your Java method uses, run the javap command as follows:

```
javap -s Class
```

The method signature used is displayed as a comment after each method declaration as shown here:

```
bash# javap -s ArrayHandler
Compiled from ArrayHandler.java
public class ArrayHandler extends java.lang.Object {
  java.lang.String arrayResults[];
   /*   [Ljava/lang/String;   */
  static {};
   /*   ()V   */
  public ArrayHandler();
   /*   ()V   */
  public void displayArray();
   /*   ()V   */
  public static void main(java.lang.String[]);
   /*   ([Ljava/lang/String;)V   */
  public native void returnArray();
   /*   ()V   */
  public void sendArrayResults(java.lang.String[]);
   /*   ([Ljava/lang/String;)V   */
}
```

Use the GetMethodID function to call instance methods in an object instance, or use the Get-StaticMethodID function to call static method. Their argument lists are the same.

Call the Methods

Third, the matching instance method is called using a Call<type>Method function. The type value can be Void, Object, Boolean, Byte, Char, Short, Int, Long, Float, or Double.

The parameters to the method can be passed as a comma-separated list, an array of values to the Call<type>MethodA function, or as a va_list. The va_list is a construct often used for variable argument lists in C. Call<*type*>MethodV is the function used to pass a va_list ().

Static methods are called in a similar way except the method naming includes an additional Static identifier, CallStaticByteMethodA, and the jclass value is used instead of jobject.

The next example returns the object array by calling the sendArrayResults method from the ArrayHandler class.

```
        // ArrayHandler.java
        public class ArrayHandler {
          private String arrayResults[];
          int arraySize=-1;

          public native void returnArray();
          static{
//API Ref      :static void loadLibrary(String libraryname)
            System.loadLibrary("nativelib");
          }
          public void sendArrayResults(String results[]) {
            arraySize=results.length;
            arrayResults=new String[arraySize];
            System.arraycopy(results,0,arrayResults,0,arraySize);
```

```
  }
  public void displayArray() {
    for (int i=0; i<arraySize; i++) {
      System.out.println("array element "+i+ "= " + arrayResults[i]);
    }
  }
  public static void main(String args[]) {
    String ar[];
    ArrayHandler ah= new ArrayHandler();
    ah.returnArray();
    ah.displayArray();
  }
}
```

The native C++ code is defined as follows:

```
// file: nativelib.cc
#include <jni.h>
#include <iostream.h>
#include "ArrayHandler.h"

JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jobject jobj){
  jobjectArray ret;
  int i;
  jclass cls;
  jmethodID mid;
  char *message[5]= {"first", "second", "third", "fourth", "fifth"};
//API Ref     :jarray NewObjectArray(JNIEnv *env, jsize length, jclass elementClass, jobject initialElement)
  ret=(jobjectArray)env->NewObjectArray(5,
      env->FindClass("java/lang/String"),
      env->NewStringUTF(""));
  for(i=0;i<5;i++) {
    env->SetObjectArrayElement(ret,i,env->NewStringUTF(message[i]));
  }
  cls=env->GetObjectClass(jobj);
  mid=env->GetMethodID(cls, "sendArrayResults", "([Ljava/lang/String;)V");
  if (mid == 0) {
    cout <<“Can't find method sendArrayResults";
    return;
  }
  env->ExceptionClear();
  env->CallVoidMethod(jobj, mid, ret);
  if(env->ExceptionOccurred()) {
    cout << "error occured copying array back" <<endl;
    env->ExceptionDescribe();
    env->ExceptionClear();
  }
  return;
}
```

To build this on Linux, run the following commands:

```
javac ArrayHandler.java
javah -jni ArrayHandler

g++  -o libnativelib.so -shared -Wl,-soname,libnative.so -I/export/home/jdk1.2/include -I/export/home/jdk1.2/include/
linux nativelib.cc  -lc
```

If you want to specify a super class method to, for example, call the parent constructor, you can do so by calling the CallNonvirtual<*type*>Method functions. One important point when calling Java methods or fields from within native code is you need to catch any raised exceptions. The ExceptionClear function clears any pending exceptions while the ExceptionOccured function checks to see if an exception has been raised in the current JNI session.

Accessing Fields

Accessing Java fields from within native code is similar to calling Java methods. However, the set or field is retrieved with a field ID, instead of a method ID.

The first thing you need to do is retrieve a field ID. You can use the GetFieldID function, but specify the field name and signature in place of the method name and signature. Once you have the field ID, call a Get<type>Field function to set the field value. The <type> is the same as the native type being returned except the *j* is dropped and the first letter is capitalized. For example, the <type> value is Int for native type jint, and Byte for native type jbyte.

The Get<type>Field function result is returned as the native type. For example, to retrieve the arraySize field in the ArrayHandler class, call GetIntField as shown in the following example.

The field can be set by calling the env->SetIntField(jobj, fid, arraysize) functions. Static fields can be set by calling SetStaticIntField(jclass, fid, arraysize) and retrieved by calling GetStaticIntField(jclass, fid).

```
#include <jni.h>
#include <iostream.h>
#include "ArrayHandler.h"

JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jobject jobj){
   jobjectArray ret;
   int i;
   jint arraysize;
   jclass cls;
   jmethodID mid;
   jfieldID fid;
   char *message[5]= {"first", "second", "third", "fourth", "fifth"};

   ret=(jobjectArray)env->NewObjectArray(5,
      env->FindClass("java/lang/String"),
      env->NewStringUTF(""));
   for(i=0;i<5;i++) {
      env->SetObjectArrayElement(ret,i,env->NewStringUTF(message[i]));
    }
    cls=env->GetObjectClass(jobj);
    mid=env->GetMethodID(cls, "sendArrayResults", "([Ljava/lang/String;)V");
    if (mid == 0) {
        cout <<"Can't find method sendArrayResults";
        return;
    }
    env->ExceptionClear();
    env->CallVoidMethod(jobj, mid, ret);
    if(env->ExceptionOccurred()) {
       cout << "error occured copying array back" << endl;
       env->ExceptionDescribe();
       env->ExceptionClear();
    }
    fid=env->GetFieldID(cls, "arraySize",  "I");
    if (fid == 0) {
        cout <<"Can't find field arraySize";
        return;
    }
    arraysize=env->GetIntField(jobj, fid);
    if(!env->ExceptionOccurred()) {
       cout<< "size=" << arraysize << endl;
    } else {
       env->ExceptionClear();
    }
    return;
}
```

Threads and Synchronization

Although the native library is loaded once per class, individual threads in an application written in the Java programming language use their own interface pointer when calling the native method. If you need to restrict access to a Java object from within native code, you can either ensure that the Java methods you call have explicit synchronization or you can use the JNI MonitorEnter and MonitorExit functions.

In the Java programming language, code is protected by a monitor whenever you specify the synchronized keyword, and the monitor enter and exit routines are normally hidden from the application developer. In JNI, you need to explicitly delineate the entry and exit points of thread safe code.

The following example uses a Boolean object to restrict access to the CallVoidMethod function.

```
env->ExceptionClear();
env->MonitorEnter(lock);
env->CallVoidMethod(jobj, mid, ret);
env->MonitorExit(lock);
if(env->ExceptionOccurred()) {
    cout << "error occured copying array back" << endl;
    env->ExceptionDescribe();
    env->ExceptionClear();
  }
```

You may find that in cases where you want access to a local system resource like an MFC window handle or message queue, it is better to use one Java.lang.Thread and access the local threaded native event queue or messaging system from within the native code.

Memory Issues

By default, JNI uses local references when creating objects inside a native method. This means when the method returns, the references are eligible to be garbage collected. If you want an object to persist across native method calls, use a global reference instead. A global reference is created from a local reference by calling NewGlobalReference on the local reference.

You can explicitly mark a reference for garbage collection by calling DeleteGlobalRef on the reference. You can also create a weak style Global reference that is accessible outside the method, but can be garbage collected. To create one of these references, call NewWeakGlobalRef and DeleteWeakGlobalRef to mark the reference for garbage collection.

You can even explicitly mark a local reference for garbage collection by calling the env->DeleteLocalRef(localobject) method. This is useful if you are using a large amount of temporary data.

```
        static jobject stringarray=0;

        JNIEXPORT void JNICALL Java_ArrayHandler_returnArray(JNIEnv *env, jobject jobj){
          jobjectArray ret;
          int i;
          jint arraysize;
          int asize;
          jclass cls, tmpcls;
          jmethodID mid;
          jfieldID fid;
          char *message[5]= {"first", "second", "third", "fourth", "fifth"};
          ret=(jobjectArray)env->NewObjectArray(5,
              env->FindClass("java/lang/String"),
              env->NewStringUTF(""));
         //Make the array available globally
//API Ref      :jobject NewGlobalRef(JNIEnv *env, jobject object)
          stringarray=env->NewGlobalRef(ret);
         //Process array
         // ...
         //clear local reference when finished..
//API Ref      :void DeleteLocalRef(JNIEnv *env, jobject localref)
          env->DeleteLocalRef(ret);
         }
```

Invocation

The section on calling methods showed you how to call a method or field in a Java program using the JNI interface and a class loaded using the FindClass function. With a little more code, you can create a standalone program that invokes a Java virtual machine and includes its own JNI interface pointer that can be used to create instances of Java classes. In the Java 2 release, the runtime program named java is a small JNI application that does exactly that.

You can create a Java virtual machine with a call to JNI_CreateJavaVM, and shut the created Java virtual machine down with a call to JNI_DestroyJavaVM. A Java virtual machine might also need some additional environment properties. These properties can be passed to the JNI_CreateJavaVM function in a JavaVMInitArgs structure.

The JavaVMInitArgs structure contains a pointer to a JavaVMOption value used to store environment information such as the classpath and Java virtual machine version, or system properties that would normally be passed on the command line to the program.

When the JNI_CreateJavaVM function returns, you can call methods and create instances of classes using the FindClass and NewObject functions the same way you would for embedded native code.

Note: The Java virtual machine invocation was only used for native thread Java virtual machines. Some older Java virtual machines have a green threads option that is stable for invocation use. On a Unix platform, you may also need to explicitly link with -lthread or -lpthread.

This next program invokes a Java virtual machine, loads the ArrayHandler class, and retrieves the arraySize field which should contain the value minus one. The Java virtual machine options include the current path in the classpath and turning the Just-In-Time (JIT) compiler off with the option -Djava.compiler=NONE.

```
#include <jni.h>

void main(int argc, char *argv[], char **envp) {
  JavaVMOption options[2];
  JavaVMInitArgs vm_args;
  JavaVM *jvm;
  JNIEnv *env;
  long result;
  jmethodID mid;
  jfieldID fid;
  jobject jobj;
  jclass cls;
  int i, asize;

  options[0].optionString = ".";
  options[1].optionString = "-Djava.compiler=NONE";
```

```
     vm_args.version = JNI_VERSION_1_2;
     vm_args.options = options;
     vm_args.nOptions = 2;
     vm_args.ignoreUnrecognized = JNI_FALSE;
//API Ref     :jint JNI_CreateJavaVM(JavaVM **pvm, void **penv, void *args)
     result = JNI_CreateJavaVM(&jvm,(void **)&env, &vm_args);
     if(result == JNI_ERR ) {
       printf("Error invoking the JVM");
       exit (-1);
     }
     cls = (*env)->FindClass(env,"ArrayHandler");
     if( cls == NULL ) {
       printf("can't find class ArrayHandler\n");
       exit (-1);
     }
     (*env)->ExceptionClear(env);
     mid=(*env)->GetMethodID(env, cls, "<init>", "()V");
     jobj=(*env)->NewObject(env, cls, mid);
     fid=(*env)->GetFieldID(env, cls, "arraySize", "I");
     asize=(*env)->GetIntField(env, jobj, fid);
     printf("size of array is %d",asize);
     (*jvm)->DestroyJavaVM(jvm);
    }
```

Attaching Threads

After the Java virtual machine is invoked, there is one local thread running the Java virtual machine. You can create more threads in the local operating system and attach the Java virtual machine to those new threads. You might want to do this if your native application is multithreaded.

Attach the local thread to the Java virtual machine with a call to AttachCurrentThread. You need to supply pointers to the Java virtual machine instance and JNI environment. In the Java 2 platform, you can also specify in the third parameter the thread name and/or group you want this new thread to live under. It is important to detach any thread that has been previously attached; otherwise, the program will not exit when you call DestroyJavaVM.

```
#include <jni.h>
#include <pthread.h>

JavaVM *jvm;

void *native_thread(void *arg) {
  JNIEnv *env;
  jclass cls;
  jmethodID mid;
  jfieldID fid;
  jint result;
  jobject jobj;
  JavaVMAttachArgs args;
  jint asize;

  args.version= JNI_VERSION_1_2;
  args.name="user";
  args.group=NULL;
  result=(*jvm)->AttachCurrentThread(jvm, (void **)&env, &args);
  cls = (*env)->FindClass(env,"ArrayHandler");
  if( cls == NULL ) {
    printf("can't find class ArrayHandler\n");
    exit (-1);
  }
  (*env)->ExceptionClear(env);
  mid=(*env)->GetMethodID(env, cls, "<init>", "()V");
  jobj=(*env)->NewObject(env, cls, mid);
  fid=(*env)->GetFieldID(env, cls, "arraySize", "I");
  asize=(*env)->GetIntField(env, jobj, fid);
  printf("size of array is %d\n",asize);
  (*jvm)->DetachCurrentThread(jvm);
}
void main(int argc, char *argv[], char **envp) {
  JavaVMOption *options;
  JavaVMInitArgs vm_args;
  JNIEnv *env;
  jint result;
  pthread_t tid;
  int thr_id;
  int i;
  options = (void *)malloc(3 * sizeof(JavaVMOption));
  options[0].optionString = "-Djava.class.path=.";
  options[1].optionString = "-Djava.compiler=NONE";
  vm_args.version = JNI_VERSION_1_2;
  vm_args.options = options;
  vm_args.nOptions = 2;
  vm_args.ignoreUnrecognized = JNI_FALSE;
  result = JNI_CreateJavaVM(&jvm,(void **)&env, &vm_args);
  if(result == JNI_ERR ) {
    printf("Error invoking the JVM");
    exit (-1);
  }
  thr_id=pthread_create(&tid, NULL, native_thread, NULL);
// If you don't have join, sleep instead
// sleep(1000);
  pthread_join(tid, NULL);
  (*jvm)->DestroyJavaVM(jvm);
  exit(0);
}
```