

Java interface programming: Everything off the best website I've ever found:

<http://www-lia.deis.unibo.it/Misc/SW/Java/tutorial/html/native1.1/implementing/index.html>

Table of Contents

Java Native Interface Programming

The JDK1.1 supports the Java Native Interface (JNI). On one hand, the JNI defines a standard naming and calling convention so that the Java Virtual Machine (VM) can locate and invoke your native methods. On the other hand, the JNI offers a set of standard interface functions. You call JNI functions from your native method code to do such things as access and manipulate

Java objects, release Java objects, create new objects, call Java methods, and so on.

This section shows you how to follow the JNI naming and calling conventions, and how to use JNI functions from a native method. Each example consists of a Java program that calls various native methods implemented in C. The native methods, in turn, may call JNI functions to access the Java objects.

The section also shows you how to embed the Java Virtual Machine into a native application.

Declaring Native Methods

On the Java side, you declare a native method with the native keyword and an empty method body. On the native language side, you provide an implementation for the native method. You must take care when writing native methods to "match" the native function implementation with the method signature in Java. javah, explained in Step 3: Create the .h file, is a helpful tool to generate native function prototypes that match the Java native method declaration.

Mapping between Java and Native Types

The JNI defines a mapping of Java types and native (C/C++) types. This section introduces the native types corresponding to both primitive Java types (e.g., int, double) and Java objects (including strings and arrays).

Accessing Java Strings

Strings are a particularly useful kind of Java objects. The JNI provides a set of string manipulation functions to ease the task of handling Java strings in native code. The programmer can translate between Java strings and native strings in Unicode and UTF-8 formats.

Accessing Java Arrays

Arrays are another kind of frequently-used Java object. You can use JNI array manipulation functions to create arrays and access array elements.

Calling Java Methods

The JNI supports a complete set of "callback" operations that allow you to invoke a Java method from the native code. You locate the method using its name and signature. You can invoke both static and instance (non-static) methods. javap is a helpful tool to generate JNI-style method signatures from class files.

Accessing Java Fields

The JNI allows you to locate the field using the field's name and type signature. You can get hold of both static and instance (non-static) fields. javap is a helpful tool to generate JNI-style field signatures from class files.

Catching and Throwing Exceptions

This section teaches you how to deal with exceptions from within a native method implementation. Your native method can catch, throw, and clear exceptions.

Local and Global References

Native code can refer to Java objects using either local or global references. Local references are only valid within a native method invocation. They are freed automatically after the native method returns. You must explicitly allocate and free global references.

Threads and Native Methods

This section describes the implications of running native methods in the multi-threaded Java environment. The JNI offers basic synchronization constructs for native methods.

Invoking the Java Virtual Machine

This section shows you how to load the Java Virtual Machine from a native library into a native application. This lesson includes how to initialize the Java Virtual Machine and invoke Java methods. The Invocation API also allows native threads to attach to a running Java Virtual Machine and bootstrap themselves into Java threads. Currently, the JDK only supports attaching native threads on Win32. The support for Solaris native threads will be available in a future release.

JNI Programming in C++

In C++, the JNI presents a slightly cleaner interface and performs additional static type checking.

Declaring Native Methods

This page shows you how to declare a native method in Java and how to generate the C/C++ function prototype.

The Java Side

Our first example, `Prompt.java`, contains a native method that takes and prints a Java string, waits for user input, and then returns the line that the user typed in.

The Java class `Prompt` contains a main method which is used to invoke the program. In addition, there is a `getLine` native method:

```
private native String getLine(String prompt);
```

Notice that the declarations for native methods are almost identical to the declarations for regular, non-native Java methods. There are two differences. Native methods must have the `native` keyword. The `native` keyword informs the Java compiler that the implementation for this method is provided in another language. Also, the native method declaration is terminated with a

semicolon, the statement terminator symbol, because there are no implementations for native methods in the Java class file.

The Native Language Side

You must declare and implement native methods in a native language, such as C or C++. Before you do this, it is helpful to generate the header file that contains the function prototype for the native method implementation.

First, compile the `Prompt.java` file and then generate the `.h` file. Compile the `Prompt.java` file as follows:

```
javac Prompt.java
```

Once you have successfully compiled `Prompt.java` and have created the `Prompt.class` file, you can generate a JNI-style header file by specifying a `-jni` option to `javah`:

```
javah -jni Prompt
```

Examine the `Prompt.h` file. Note the function prototype for the native method `getLine` that you declared in `Prompt.java`.

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

The native method function definition in the implementation code must match the generated function prototype in the header file.

Always include `JNIEXPORT` and `JNICALL` in your native method function prototypes. `JNIEXPORT` and `JNICALL` ensure that the source code compiles on platforms such as Win32 that require special keywords for functions exported from dynamic link libraries.

Native method names are concatenated from the following components:

- the prefix `Java_`
- the fully qualified class name
- an underscore `"_"` separator
- the method name

(Note that overloaded native method names, in addition to the above components, have extra two underscores `"__"` appended to the method name followed by the argument signature.)

As a result, the `Prompt.getLine` method is implemented by `Java_Prompt_getLine` in native code. (There is no package name component because the `Prompt` class is in the default package.)

Each native method has two additional parameters, in addition to any parameters that you declare on the Java side. The first parameter, `JNIEnv *`, is the JNI interface pointer. This interface pointer is organized as a function table, with every JNI function at a known table entry. Your native method invokes specific JNI functions to access Java objects through the `JNIEnv`

`*` pointer. The `jobject` parameter is a reference to the object itself (it is like the `this` pointer in C++).

Lastly, notice that JNI has a set of type names (e.g., `jobject`, `jstring`) that correspond to Java types.

Mapping between Java and Native Types

In this section, you will learn how to reference Java types in your native method. This is useful when you want to access the arguments passed in from Java, create new Java objects, and return results to the caller.

Java Primitive Types

Your native method can directly access Java primitive types such as booleans, integers, floats, and so on, that are passed from Java programs. For example, the Java type `boolean` maps to the native type `jboolean` (represented as unsigned 8 bits), while the Java type `float` maps to the native type `jfloat` (represented by 32 bits). The following table describes the mapping of Java primitive types to native types.

Java Primitive Types and Native Equivalents

Java Type	Native Type	Size in bits
<code>boolean</code>	<code>jboolean</code>	8, unsigned
<code>byte</code>	<code>jbyte</code>	8
<code>char</code>	<code>jchar</code>	16, unsigned
<code>short</code>	<code>jshort</code>	16
<code>int</code>	<code>jint</code>	32
<code>long</code>	<code>jlong</code>	64
<code>float</code>	<code>jfloat</code>	32
<code>double</code>	<code>jdouble</code>	64
<code>void</code>	<code>void</code>	n/a

Java Object Types

Java objects are passed by reference. All references to Java objects have type `jobject`. For convenience and to reduce the chance of programming errors, we introduce a set of types that are conceptually all "subtypes" of `jobject`:

- `jobject` represents all Java objects.
- `jclass` represents Java class objects (`java.lang.Class`).
- `jstring` represents Java strings (`java.lang.String`).
- `jarray` represents Java arrays.
 - `jobjectArray` represents arrays of objects.
 - `jbooleanArray` represents boolean arrays.
 - `jbyteArray` represents byte arrays.
 - `jcharArray` represents char arrays.

jshortArray represents short arrays.
jintArray represents int arrays.
jlongArray represents long arrays.
jfloatArray represents float arrays.
jdoubleArray represents double arrays.
jthrowable represents Java exceptions (java.lang.Throwable).

In our Prompt.java example, the native method `getLine`:

```
private native String getLine(String prompt);
```

takes a Java string as an argument and returns a Java string. Its corresponding native implementation has type:

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

As mentioned above, `jstring` corresponds to Java type `String`. The second argument, which is the reference to the object itself, has type `jobject`.

Java Strings

The `jstring` type passed from Java to native code is not the regular C string type (`char *`). Therefore, trying to directly print a `jstring` will likely result in a VM crash.

```
/* DO NOT USE jstring THIS WAY !!! */
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    printf("%s", prompt);
}
```

Your native method code must use JNI functions to convert Java strings to native strings. The JNI supports the conversion to and from native Unicode and UTF-8 strings. In particular, UTF-8 strings use the highest bit to signal multibyte characters; they are therefore upward compatible with 7-bit ASCII. In Java, UTF-8 strings are always 0-terminated.

Accessing Java Strings

To correctly print the string passed in from Java, your native method needs to call `GetStringUTFChars`. `GetStringUTFChars` converts the built-in unicode representation of a Java string into a UTF-8 string. If you are certain that the string only contains 7-bit ASCII characters, you can directly pass the string to regular C functions, such as `printf`, as is shown in `Prompt.c`.

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
    char buf[128];
    const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
    printf("%s", str);
    (*env)->ReleaseStringUTFChars(env, prompt, str);
}
```

Note that when your native code is finished using the UTF-8 string, it must call `ReleaseStringUTFChars`. `ReleaseStringUTFChars` informs the VM that the native method is finished with the string, so that the memory taken by the UTF-8 string can then be freed. Failing to call `ReleaseStringUTFChars` results in a memory leak. This will ultimately lead to system memory exhaustion.

The native method can also construct a new string using the JNI function `NewStringUTF`. The following lines of code from `Java_Prompt_getLine` show this:

```
scanf("%s", buf);
return (*env)->NewStringUTF(env, buf);
}
```

Using the JNIEnv Interface Pointer

Native methods must access and manipulate Java objects, such as strings, through the `env` interface pointer. In C, this requires using the `env` pointer to reference the JNI function. Notice how the native method uses the `env` interface pointer to reference the two functions, `GetStringUTFChars` and `ReleaseStringUTFChars`, that it calls. In addition, `env` is passed as the first parameter to these functions.

Other JNI Functions for Accessing Java Strings

The JNI also provides functions to obtain the Unicode representation of Java strings. This is useful, for example, on those operating systems that support Unicode as the native format. There are also utility functions to obtain both the UTF-8 and Unicode length of Java strings.

`GetStringChars` takes the Java string and returns a pointer to an array of Unicode characters that comprise the string.

`ReleaseStringChars` releases the pointer to the array of Unicode characters.

`NewString` constructs a new `java.lang.String` object from an array of Unicode characters.

`GetStringLength` returns the length of a string that is comprised of an array of Unicode characters.

`GetStringUTFLength` returns the length of a string if it is represented in the UTF-8 format.

Accessing Java Arrays

Similar to `jstring`, `jarray` represents references to Java arrays and cannot be directly accessed in C. Our second example, `IntArray.java`, contains a native method that totals up the contents of an integer array. You cannot implement the native method by directly addressing the array elements:

```
/* This program is illegal! */
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
    int i, sum = 0;
    for (i=0; i<10; i++)
        sum += arr[i];
}
```

Instead, the JNI provides functions that allow you to obtain pointers to elements of integer arrays. The correct way to implement the above function is shown in the native method `IntArray.c`.

Accessing Arrays of Primitive Elements

First, obtain the length of the array by calling the JNI function `GetArrayLength`. Note that, unlike C arrays, Java arrays carry length information.

```
JNIEXPORT jint JNICALL
Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
    int i, sum = 0;
    jsize len = (*env)->GetArrayLength(env, arr);
}
```

Next, obtain a pointer to the elements of the integer array. Our example uses `GetIntArrayElements` to obtain this pointer. You can use normal C operations on the resulting integer array.

```
jint *body = (*env)->GetIntArrayElements(env, arr, 0);
for (i=0; i<len; i++)
    sum += body[i];
```

While, in general, Java arrays may be moved by the garbage collector, the Virtual Machine guarantees that the result of `GetIntArrayElements` points to a nonmovable array of integers. The JNI will either "pin" down the array, or it will make a copy of the array into nonmovable memory. When the native code has finished using the array, it must call `ReleaseIntArrayElements`, as follows:

```
(*env)->ReleaseIntArrayElements(env, arr, body, 0);
return sum;
}
```

`ReleaseIntArrayElements` enables the JNI to copy back and free `body` if it is a copy of the original Java array, or "unpin" the Java array if it has been pinned in memory. Forgetting to call `ReleaseIntArrayElements` results in either pinning the array for an extended period of time, or not being able to reclaim the memory used to store the nonmovable copy of the array.

The JNI provides a set of functions to access arrays of every primitive type, including boolean, byte, char, short, int, long, float, and double:

- `GetBooleanArrayElements` accesses elements in a Java boolean array.
- `GetByteArrayElements` accesses elements in a Java byte array.
- `GetCharArrayElements` accesses elements in a char array.
- `GetShortArrayElements` accesses elements in a short array.
- `GetIntArrayElements` accesses elements in an int array.
- `GetLongArrayElements` accesses elements in a long array.

GetFloatArrayElements accesses elements in a float array.
GetDoubleArrayElements accesses elements in a double array.

Accessing a Small Number of Elements

Note that the Get<type>ArrayElements function might result in the entire array being copied. If you are only interested in a small number of elements in a (potentially) large array, you should instead use the Get/Set<type>ArrayRegion functions. These functions allow you to access, via copying, a small set of elements in an array.

Accessing Arrays of Objects

The JNI provides a separate set of function to access elements of object arrays. You can get and set individual object array elements. You cannot get all the object array elements at once.

GetObjectArrayElement returns the object element at a given index.
SetObjectArrayElement updates the object element at a given index.

Calling Java Methods

This section illustrates how you can call Java methods from native methods. Our example program, `Callbacks.java`, invokes a native method. The native method then makes a call back to a Java method. To make things a little more interesting, the Java method again (recursively) calls the native method. This process continues until the recursion is five levels deep, at which time the Java method returns without making any more calls to the native method. To help you see this, the Java method and the native method print a sequence of tracing information.

Calling a Java Method from Native Code

Let us focus on the implementation of `Callbacks_nativeMethod`, implemented in `Callbacks.c`. This native method contains a call back to the Java method `Callbacks.callback`.

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "(I)V");
    if (mid == 0)
        return;
    printf("In C, depth = %d, about to enter Java\n", depth);
    (*env)->CallVoidMethod(env, obj, mid, depth);
    printf("In C, depth = %d, back from Java\n", depth);
}
```

You can call an instance (non-static) method by following these three steps:

Your native method calls `GetObjectClass`. This returns the Java class object to which the Java object belongs. Your native method then calls `GetMethodID`. This performs a lookup for the Java method in a given class. The lookup is based on the name of the method as well as the method signature. If the method does not exist, `GetMethodID` returns 0. An immediate return from the native method at that point causes a `NoSuchMethodError` to be thrown in Java code. Lastly, your native method calls `CallVoidMethod`. This invokes an instance method that has void return type. You pass the object, method ID, and the actual arguments to `CallVoidMethod`.

Forming the Method Name and Method Signature

The JNI performs a symbolic lookup based on the method's name and type signature. This ensures that the same native method will work even after new methods have been added to the corresponding Java class.

The method name is the Java method name in UTF-8 form. Specify the method name for a constructor of a class by enclosing the word `init` within angle brackets (this appears as "`<init>`").

Note that the JNI uses method signatures to denote the type of Java methods. The signature `(I)V`, for example, denotes a Java method that takes one argument of type `int` and has return type `void`. The general form of a method signature argument is:

`"(argument-types)return-type"`

The following table summarizes the encoding for the Java type signatures:

Java VM Type Signatures

Signature	Java Type
Z	

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

For example, the `Prompt.getLine` method has the signature:

```
(Ljava/lang/String;)Ljava/lang/String;
```

whereas the `Callbacks.main` method has the signature:

```
([Ljava/lang/String;)V
```

Array types are indicated by a leading square bracket (`[]`) followed by the type of the array elements.

Using `javap` to Generate Method Signatures

To eliminate the mistakes in deriving method signatures by hand, you can use the `javap` tool to print out method signatures. For example, by running:

```
javap -s -p Prompt
```

you can obtain the following output:

```
Compiled from Prompt.java
class Prompt extends java.lang.Object
  /* ACC_SUPER bit set */
  {
    private native getLine (Ljava/lang/String;)Ljava/lang/String;
    public static main ([ Ljava/lang/String;)V
    <init> ()V
    static <clinit> ()V
  }
```

The `"-s"` flag informs `javap` to output signatures rather than normal Java types. The `"-p"` flag causes private members to be included.

Calling Java Methods Using Method IDs

In JNI, you pass the method ID to the actual method invocation function. This makes it possible to first obtain the method ID, which is a relatively expensive operation, and then use the method ID many times at later points to invoke the same method.

It is important to keep in mind that a method ID is valid only as long as the class from which it is derived is not unloaded. Once the class is unloaded, the method ID becomes invalid. So if you want to cache the method ID, make sure to keep a live reference to the Java class from which the method ID is derived. As long as the reference to the Java class (the jclass value) exists, the native code keeps a live reference to the class. The section Local and Global References explains how to keep a live reference even after the native method returns and the jclass value goes out of scope.

Passing Arguments to Java Methods

The JNI provides several ways to pass arguments to a Java method. Most often, you pass the arguments following the method

ID. There are also two variations of method invocation functions that take arguments in an alternative format. For example, the

CallVoidMethodV function receives the arguments in a va_list, and the CallVoidMethodA function expects the arguments in an array of jvalue union types:

```
typedef union jvalue {
    jboolean z;
    jbyte b;
    jchar c;
    jshort s;
    jint i;
    jlong j;
    jfloat f;
    jdouble d;
    jobject l;
} jvalue;
```

Besides CallVoidMethod function, the JNI also supports instance method invocation functions with other return types, such as CallBooleanMethod, CallIntMethod, and so on. The return type of the method invocation function must match with the Java method you wish to invoke.

Calling Static Methods

You can call static Java method from your native code by following these steps:

Obtain the method ID using GetStaticMethodID, as opposed to GetMethodID.

Pass the class, method ID, and arguments to the family of static method invocation functions: CallStaticVoidMethod, CallStaticBooleanMethod, and so on.

If you compare instance method invocation functions to static method invocation functions, you will notice that instance method

invocation functions receive the object, rather than the class, as the second argument following the JNIEnv argument. For example, if we add a static method

```
static int incDepth(int depth) {return depth + 1};
```

into Callback.java, we can call it from Java_Callback_nativeMethod using:

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
    jclass cls = (*env)->GetObjectClass(env, obj);
    jmethodID mid = (*env)->GetStaticMethodID(env, cls, "incDepth", "(I)I");
    if (mid == 0)
        return;
    depth = (*env)->CallStaticIntMethod(env, cls, mid, depth);
}
```

Calling Instance Methods of a Superclass

You can call instance methods defined in a superclass that have been overridden in the class to which the object belongs. The

JNI provides a set of `CallNonvirtual<type>Method` functions for this purpose. To call instance methods from the superclass that defined them, you do the following:

Obtain the method ID from the superclass using `GetMethodID`, as opposed to `GetStaticMethodID`).

Pass the object, superclass, method ID, and arguments to the family of nonvirtual invocation functions:

`CallNonvirtualVoidMethod`, `CallNonvirtualBooleanMethod`, and so on.

It is rare that you will need to invoke the instance methods of a superclass. This facility is similar to calling a superclass method,

say `f`, using:

```
super.f();
```

in Java.

Accessing Java Fields

The JNI provides functions that native methods use to get and set Java fields. You can get and set both instance fields and static (class) fields. Similar to accessing Java methods, you use different JNI functions to access instance and static fields.

Our example program, `FieldAccess.java`, contains a class with one static integer field `si` and an instance string field `s`. It calls the native method `accessFields`, which prints out the value of these two fields, and then sets the fields to new values. To verify the fields have indeed changed, we print out their values again in Java after returning from the native method.

Procedure for Accessing a Java Field

To get and set Java fields from a native method, you must do the following:

Obtain the identifier for that field from its class, name, and type signature. For example, in `FieldAccess.c`, we have:

```
fid = (*env)->GetStaticFieldID(env, cls, "si", "I");
```

and:

```
fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;");
```

Use one of several JNI functions to either get or set the field specified by the field identifier. Pass the class to the appropriate static field access functions. Pass the object to the appropriate instance field access functions. For example, in `FieldAccess.c`, we have:

```
si = (*env)->GetStaticIntField(env, cls, fid);
```

and:

```
jstr = (*env)->GetObjectField(env, obj, fid);
```

Similar to calling a Java method, we factor out the cost of field lookup using a two-step process. The field ID uniquely identifies a field in a given class. Similar to method IDs, a field ID remains valid until the class from which it is derived is unloaded.

Field Signatures

Field signatures are specified following the same encoding scheme as method signatures. The general form of a field signature is:

```
"field type"
```

The field signature is the encoded symbol for the type of the field, enclosed in double quotes (""). The field symbols are the same as the argument symbols in the method signature. That is, you represent an integer field with "I", a float field with "F", a double field with "D", a boolean field with "Z", and so on.

The signature for a Java object, such as a `String`, begins with the letter `L`, followed by the fully-qualified class for the object, and terminated by a semicolon (;). Thus, you form the field signature for a `String` variable (`c.s` in `FieldAccess.java`) as follows:

```
"Ljava/lang/String;"
```

Arrays are indicated by a leading square bracket ([]) followed by the type of the array. For example, you designate an integer array as follows:

"[I"

Refer to the table in previous section which summarizes the encoding for the Java type signatures and their matching Java types.

You can use javap with option "-s" to generate the field signatures from class files. For example, run:

```
javap -s -p FieldAccess
```

This gives you output containing:

```
...  
static si I  
s Ljava/lang/String;  
...
```

Catching and Throwing Exceptions

In Java, once an exception is thrown, the Virtual Machine automatically looks for the nearest enclosing exception handler, and unwinds the stack if necessary. The advantage of this style of exceptions is that the programmer does not have to care about unusual error cases for every operation in the program. Instead, the error conditions are propagated automatically to a location (the catch clause in Java) where the same class of error conditions can be handled in a centralized way.

Although certain languages such as C++ support a similar notion of exception handling, there is no uniform and general way to throw and catch exceptions in native languages. The JNI therefore requires you to check for possible exceptions after calling JNI functions. The JNI also provides functions to throw Java exceptions, which can then be handled either by other parts of your native code, or by the Java Virtual Machine. After the native code catches and handles an exception, it can either clear the pending exception so that the computation may continue, or it can throw another exception for an outer exception handler.

Many JNI functions may cause an exception to be thrown. For example, the `GetFieldID` function described in the previous section throws a `NoSuchFieldError` if the specified field does not exist. To simplify error checking, most JNI functions use a combination of error codes and Java exceptions to report error conditions. You may, for example, check if the `jfieldID` returned from `GetFieldID` is 0, instead of calling the JNI function `ExceptionOccurred`. When the result of `GetFieldID` is not 0, it is guaranteed that there is no pending exception.

The remainder of the section illustrates how to catch and throw exceptions in native code. The example code is in `CatchThrow.java`.

The `CatchThrow.main` method calls the native method. The native method, defined in `CatchThrow.c`, first invokes the Java method `CatchThrow.callback`:

```
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "()V");
jthrowable exc;
if (mid == 0)
    return;
(*env)->CallVoidMethod(env, obj, mid);
```

Note that since `CallVoidMethod` throws a `NullPointerException`, the native code can detect this exception after `CallVoidMethod` returns by calling `ExceptionOccurred`:

```
exc = (*env)->ExceptionOccurred(env);
if (exc) {
```

This is how you can catch and handle an exception. In our example, we do not do much about the exception in `CatchThrow.c`, except use `ExceptionDescribe` to output some debugging message. It then throws an `IllegalArgumentException`. It is this `IllegalArgumentException` that the Java code which invoked the native method will see.

```
(*env)->ExceptionDescribe(env);
(*env)->ExceptionClear(env);

newExcCls = (*env)->FindClass(env, "java/lang/IllegalArgumentException");
if (newExcCls == 0) /* Unable to find the new exception class, give up. */
    return;
(*env)->ThrowNew(env, newExcCls, "thrown from C code");
```

`ThrowNew` constructs an exception object from the given exception class and message string and posts the exception in the current thread.

Note that it is extremely important to check, handle, and clear the pending exception before we call the subsequent JNI

functions. Calling arbitrary JNI functions with a pending exception may lead to unexpected results. Only a small number of JNI functions are safe to call when there is a pending exception. They include `ExceptionOccurred`, `ExceptionDescribe`, and `ExceptionClear`.

Local and Global References

So far, we have used data types such as `jobject`, `jclass`, and `jstring` to denote references to Java objects. The JNI creates references for all object arguments passed in to native methods, as well as all objects returned from JNI functions.

These references will keep the Java objects from being garbage collected. To make sure that Java objects can eventually be freed, the JNI by default creates local references. Local references become invalid when the execution returns from the native method in which the local reference is created. Therefore, a native method must not store away a local reference and expect to reuse it in subsequent invocations.

For example, the following program (a variation of the native method in `FieldAccess.c`) mistakenly caches the Java class for the field ID so that it does not have to repeatedly search for the field ID based on the field name and signature:

```
/* This code is illegal */
static jclass cls = 0;
static jfieldID fld;

JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
    ...
    if (cls == 0) {
        cls = (*env)->GetObjectClass(env, obj);
        if (cls == 0)
            ... /* error */
        fld = (*env)->GetStaticFieldID(env, cls, "si", "I");
    }
    ... /* access the field using cls and fld */
}
```

This program is illegal because the local reference returned from `GetObjectClass` is only valid before the native method returns. When `Java_FieldAccess_accessField` is entered the second time, an invalid local reference will be used. This leads to wrong results or to a VM crash.

To overcome this problem, you need to create a global reference. This global reference will remain valid until it is explicitly freed:

```
/* This code is OK */
static jclass cls = 0;
static jfieldID fld;

JNIEXPORT void JNICALL
Java_FieldAccess_accessFields(JNIEnv *env, jobject obj)
{
    ...
    if (cls == 0) {
        jclass cls1 = (*env)->GetObjectClass(env, obj);
        if (cls1 == 0)
            ... /* error */
        cls = (*env)->NewGlobalRef(env, cls1);
        if (cls == 0)
            ... /* error */
        fld = (*env)->GetStaticFieldID(env, cls, "si", "I");
    }
    ... /* access the field using cls and fld */
}
```

A global reference keeps the Java class from being unloaded, and therefore also ensures that the field ID remains valid, as discussed in *Accessing Java Fields*. The native code must call `DeleteGlobalRefs` when it no longer needs access to the global reference; otherwise, the corresponding Java object (e.g., the Java class referenced to by `cls` above) will never be unloaded.

In most cases, the native programmer should rely on the VM to free all local references after the native method returns. In certain situations, however, the native code may need to call the `DeleteLocalRef` function to explicitly delete a local reference. These situations are:

You may know that you are holding the only reference to a large Java object, and you do not want to wait until the current native method returns before the Java object can be reclaimed by the garbage collector. For example, in the following program segment, the garbage collector may be able to free the Java object referred to by `lref` when it is running inside `lengthyComputation`:

```
lref = ...          /* a large Java object */

...                /* last use of lref */
(*env)->DeleteLocalRef(env, lref);

lengthyComputation(); /* may take some time */

return;           /* all local refs will now be freed */
}
```

You may need to create a large number of local references in a single native method invocation. This may result in an overflow of the internal JNI local reference table. It is a good idea to delete those local references that will not be needed. For example, in the following program segment, the native code iterates through a potentially large array `arr` consisting of Java strings. After each iteration, the local reference to the string element can be freed:

```
for(i = 0; i < len; i++) {
    jstring jstr = (*env)->GetObjectArrayElement(env, arr, i);
    ...           /* processes jstr */
    (*env)->DeleteLocalRef(env, jstr); /* no longer needs jstr */
}
```

Threads and Native Methods

Java is a multithreaded system, therefore native methods must be thread-safe programs. Unless you have extra knowledge (for example, the native method is synchronized), you must assume that there can be multiple threads of control running through a native method at any given time. Native methods therefore must not modify sensitive global variables in unprotected ways. That is, they must share and coordinate their access to variables in certain critical sections of code.

Before reading this section, you should be familiar with the concepts of threads of control and multithreaded programming. [Threads of Control](#) covers programming with threads. In particular, the page [Multithreaded Programs](#) covers issues related to writing programs that contain multiple threads, including how to synchronize them.

Threads and JNI

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

You must not pass local references from one thread to another. In particular, a local reference may become invalid before the other thread has had a chance to use it. Always convert them to global references when there is any doubt that a reference to Java object may be used by different threads.

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

Thread Synchronization in Native Methods

JNI provides two synchronization functions to allow you to implement synchronized blocks. In Java, they are implemented using the `synchronized` statement. For example:

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

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

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

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

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

Wait and Notify

Another useful thread synchronization mechanism is `Object.wait`, `Object.notify`, and `Object.notifyAll`. The JNI does not directly support these functions. However, a native method can always follow the JNI method call mechanism to

invoke these methods.

Invoking the Java Virtual Machine

In JDK1.1, the Java Virtual Machine is shipped as a shared library (or dynamic link library on Win32). You can embed the Java VM into your native application by linking the native application with the shared library. The JNI supports an Invocation

API that allows you to load, initialize, and invoke the Java Virtual Machine. Indeed, the normal way of starting the Java interpreter, `java`, is no more than a simple C program that parses the command line arguments and invoke the Java Virtual Machine through the Invocation API.

Invoking the Java Virtual Machine

As an example, we will write a C program to invoke the Java Virtual machine and call the `Prog.main` method defined in `Prog.java`:

```
public class Prog {
    public static void main(String[] args) {
        System.out.println("Hello World" + args[0]);
    }
}
```

The C code in `invoke.c` begins with a call to `JNI_GetDefaultJavaVMInitArgs` to obtain the default initialization settings (heap size, stack size, and so on). It then calls `JNI_CreateJavaVM` to load and initialize the Virtual Machine. `JNI_CreateJavaVM` fills in two return values:

`jvm` refers to the created Java Virtual Machine. It can be later used to, for example, destroy the Virtual Machine.
`env` is a JNI interface pointer that the current thread can use to access Java features, such as calling a Java method.

Note that after `JNI_CreateJavaVM` successfully returns, the current native thread has bootstrapped itself into the Java Virtual

Machine, and is therefore running just like a native method. The only difference is that there is no concept of returning to the

Java Virtual Machine. Therefore, the local references created subsequently will not be freed until you call `DestroyJavaVM`.

Once you have created the Java Virtual Machine, you can issue regular JNI calls to invoke, for example, `Prog.main`. `DestroyJavaVM` attempts to unload the Java Virtual Machine. (The JDK 1.1 Java Virtual Machine cannot be unloaded, therefore `DestroyJavaVM` always returns an error code.)

You need to compile and link `invoke.c` with Java libraries shipped with JDK1.1. On Solaris, you can use the following command to compile and link `invoke.c`:

```
cc -I<where jni.h is> -L<where libjava.so is> -ljava invoke.c
```

On Win32 with Microsoft Visual C++ 4.0, the command line is:

```
cl -I<where jni.h is> -MT invoke.c -link <where javai.lib is>\javai.lib
```

Run the resulting executable from the command line. If you get the following error message:

```
Unable to initialize threads: cannot find class java/lang/Thread
Can't create Java VM
```

This means that your `CLASSPATH` environment variable is not set up properly to include Java system classes. On the other hand, if the system complains that it cannot find either `libjava.so` (on Solaris) or `javai.dll` (on Win32), add `libjava.so` into your `LD_LIBRARY_PATH` on Solaris, or add `javai.dll` into your executable path on Win32. If the program complains that it can not find the class `Prog`, make sure the directory containing `Prog.class` is in the `CLASSPATH` as well.

If you are building a window (as opposed to a console) application on Win32, you must explicitly set the `CLASSPATH` slot in

the Virtual Machine initialization structure. Window applications do not read the environment strings like console applications do. In general, it always a good idea to specify the CLASSPATH before you call JNI_CreateJavaVM, rather than relying on the environment variable settings when the program is run.

Attaching Native Threads

The Invocation API also allows you to attach native threads to a running Java VM, and bootstrap themselves into Java threads. This requires that the Java Virtual Machine internally uses native threads. In JDK 1.1, this feature only works on Win32. The Solaris version of Java Virtual Machine uses user-level thread support and is therefore incapable of attaching native threads. A future version of JDK on Solaris will support native threads.

Our example program, attach.c, therefore, will only work on Win32. It is a variation of invoke.c. Instead of calling Prog.main in the main thread, the native code spawns five threads, and then just waits for them to finish before it destroys the Java Virtual Machine. Each thread attaches itself to the Java Virtual Machine, invokes the Prog.main method, and finally detaches itself from the Virtual Machine before it terminates.

All local references belonging to the current thread will be freed when DetachCurrentThread is called.

Limitations of the Invocation API in JDK1.1

As mentioned above, there are a number of limitations of the Invocation API implementation in JDK1.1.

The user-level Java thread implementation on Solaris requires the Java VM to redirect certain Solaris system calls. The set of redirected system calls currently includes read, readv, write, writev, getmsg, putmsg, poll, open, close, pipe, fcntl, dup, create, accept, recv, send, and so on. This may cause undesirable effects on a hosting native application that also depends on these system calls.

You cannot attach a native thread to the user-thread based Java Virtual Machine on Solaris. AttachCurrentThread simply fails on Solaris (unless it is called from the main thread that created the Virtual Machine).

You cannot unload the Java Virtual Machine without terminating the process. The DestroyJavaVM call simply returns an error code.

These problems will be fixed in future releases of the JDK.

JNI Programming in C++

The JNI provides a slightly cleaner interface for C++ programmers. The `jni.h` file contains a set of inline C++ functions so that the native method programmer can simply write:

```
jclass cls = env->FindClass("java/lang/String");
```

instead of:

```
jclass cls = (*env)->FindClass(env, "java/lang/String");
```

The extra level of indirection on `env` and the `env` argument to `FindClass` are hidden from the programmer. The C++ compiler simply expands out the C++ member function calls to their C counterparts; therefore, the resulting code is exactly the same.

The `jni.h` file also defines a set of dummy C++ classes to enforce the subtyping relationships among different variations of `jobject` types:

```
class _jobject {};  
class _jclass : public _jobject {};  
class _jthrowable : public _jobject {};  
class _jstring : public _jobject {};  
... /* more on jarray */
```

```
typedef _jobject *jobject;  
typedef _jclass *jclass;  
typedef _jthrowable *jthrowable;  
typedef _jstring *jstring;  
... /* more on jarray */
```

The C++ compiler is therefore able to detect if you pass in, for example, a `jobject` to `GetMethodID`:

```
jmethodID GetMethodID(jclass clazz, const char *name,  
                     const char *sig);
```

because `GetMethodID` expects a `jclass`. In C, `jclass` is simply the same as `jobject`:

```
typedef jobject jclass;
```

Therefore a C compiler is not able to detect that you have mistakenly passed a `jobject` instead of `jclass`.

The added type safety in C++ does come with a small inconvenience. Recall from [Accessing Java Arrays](#) that in C, you can fetch a Java string from an array of strings and directly assign the result to a `jstring`:

```
jstring jstr = (*env)->GetObjectArrayElement(arr, i);
```

In C++, however, you need to insert an explicit conversion:

```
jstring jstr = (jstring)env->GetObjectArrayElement(arr, i);
```

because `jstring` is a subtype of `jobject`, the return type of `GetObjectArrayElement`.