**--------------------------------------------------------------------------**

**Making Native Windows API calls from within a Java Application**

One of the main points of Java is to be completely platform independent.  However, sometimes it will occur that the developer of an application will know that his or her application is only going to be run on a specific platform, for example, Win32.

**NOTE:**  This example assumes that you are using JDK 1.2 or later.

Below are the steps for writing a Java application that makes a Win32 API call.  The application generates a Swing Jframe and makes it system modal, or gives it the Always On Top functionality, similar to that of the Windows NT Task Manager.   
-----

Steps to follow:

**1.  Write the Java code for the application**

**2. Run javah.exe on your .class file to generate a C header file**

**3. write the implementation of your native methods**

**4. create the shared library**

**5. run the application**   
-----

1.  Write the Java code for the application   
 

import java.awt.\*;   
import sun.awt.\*;   
import sun.awt.windows.\*;   
import javax.swing.\*;   
import java.awt.event.\*;

public class Frame1 extends JFrame {

  int windowHWND = 0;   
  JButton jButton1 = new JButton();

  public Frame1() {

    //windowHWND = this.getHwnd();   
    try  {   
      jbInit();   
    }   
    catch(Exception e) {   
      e.printStackTrace();   
    }

  }

  public int getHwnd() {

  DrawingSurfaceInfo w = (DrawingSurfaceInfo) ((DrawingSurface) getPeer()).getDrawingSurfaceInfo();   
  w.lock();   
  WDrawingSurfaceInfo win32 = (WDrawingSurfaceInfo) w;   
  int hwnd = win32.getHWnd();   
  w.unlock();

  return hwnd;

  }

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

  public static native void WindowAlwaysOnTop(int hwnd, boolean flag);   
    
 

  public static void main(String[] args) {   
    Frame1 frame11 = new Frame1();

    frame11.setSize(400,400);

    frame11.setVisible(true);

  }

  private void jbInit() throws Exception {

    jButton1.setText("jButton1");

    this.addWindowListener(new java.awt.event.WindowAdapter() {

      public void windowOpened(WindowEvent e) {   
        this\_windowOpened(e);   
      }

      public void windowClosing(WindowEvent e) {   
        this\_windowClosing(e);   
      }   
    });   
    this.getContentPane().add(jButton1, BorderLayout.NORTH);   
  }

  void this\_windowOpened(WindowEvent e) {

     windowHWND = this.getHwnd();   
     System.out.println("the value is: " + this.getHwnd());   
     this.WindowAlwaysOnTop(windowHWND, true);   
    
  }   
 

  void this\_windowClosing(WindowEvent e) {

    System.exit(0);

  }

}

Once the code is written, compile it with Jbuilder or the command line javac.exe tool which will result in a generated .class file.  You will use this .class file in the next step.   
-----

2. Run javah.exe on your .class file to generate a C header file

The following line represents the basic syntax for running javah.exe:

javah Frame1

where Frame1 is the name of the argument class.

When you run javah.exe, it will generate a header file by the same name as your implementation but with a .h file extension.  For this example the .h file that was generated from the above Java code will be emitted being that it is quite large.

NOTE:  Make sure that when you run javah.exe, it is the javah.exe that came with the same JDK that you will be compiling with as there may be some issues with using a version of javah.exe that is different from that of the JDK you are using to compile.   
-----

3. write the implementation of your native methods

Now that you have your Java source and your C header file, it is time to write the implementation of your native methods.

Following is the C code that corresponds to the native methods declared in the Java code listed in step 1:

#include "jni.h"   
#include "Frame1.h"   
#include <stdio.h>   
#include<windows.h>

JNIEXPORT void JNICALL Java\_Frame1\_WindowAlwaysOnTop(JNIEnv \*env, jclass obj, jint hwnd, jboolean flag)   
{   
 if (flag)   
  SetWindowPos((HWND) hwnd,HWND\_TOPMOST,0,0,0,0,SWP\_NOMOVE|SWP\_NOSIZE);   
 else   
  SetWindowPos((HWND) hwnd,HWND\_NOTOPMOST,0,0,0,0,SWP\_NOMOVE|SWP\_NOSIZE);

 return;   
}

You will notice several things: one is that the function signature has Java\_Frame1\_ preceeding the name of the function.  If there was a package statement in the Java source, it would appear after Frame1\_ in the function signature.

Second, you will notice the #include jni.h.  Normally this would be #include<jni.h>, depending on how you have your libraries set up within your C compiler.   
-----

4. create the shared library

Now you are ready to create the shared library.  Using your C compiler, create a .dll file with the code from the C implementation file.  Refer to the doccumentation of the C compiler for details on creating a .dll file.

**---------------------------------------------------------------------**   
**For those interested in using Borland C++ Builder:**

If you have got it installed, you could use Borland C++ Builder 3 or C++ Builder 4 to create your DLL file.  If this is the case, you would use **File | New... | DLL**  C++ Builder will then generate some code for you, and you will just need to add your implementation code to the code which was generated.   
**---------------------------------------------------------------------**

Remember in the Java code in step one there is a line:

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

windowOnTop is the name of the .dll file.  You can name it whatever you want, just make sure that you specify the appropriate name when loading the library.   
-----

5. run the application

Finally you are ready to run the application.  From the command line use java.exe and as the argument specify the name of the class that you compiled in step one.  Once the system loads your DLL, the window that the VM creates should mimic the functionality.

# Enhance your Java application with Java Native Interface (JNI)

### How to compete with native applications without sacrificing cross-platform benefits

By Tal Liron, JavaWorld.com, 10/20/99

So, you're going to develop a great new application. Java's cross-platform promises bring a twinkle to the eyes of your marketing people, but after a test run on a group of innocent users, you're not so sure Java can deliver the goods.

The main deficiency Java applications seem to have is that they do not look and feel like native applications. The reasons behind this are obvious: 100% Pure Java must support the lowest common denominator, but there are many platform-specific features and quirks that the users of specific platforms expect, and your application will seem to be lacking without them.

In today's competitive application arena, these features may be critically necessary to your application's marketplace success. A competitor's native application can use these features to beat your Java application every time, producing a product that's faster, looks and feels right, and works with the operating system and other native applications as expected.

The solution, of course, is to use Java Native Interface (JNI) extensions. But what will this mean for Java's cross-platform capabilities? If your application is only 60% Pure Java, it won't run on other platforms -- so why use Java at all? Admittedly, Java has other benefits: the language is easy to learn and use, the code is simple to maintain and debug, and by using it you leave open the possibility of migrating to other platforms in the future -- if you design your application with such migration in mind.

So, is JNI a compromise? It doesn't have to be. You can have a 100% Pure Java application in which JNI features are added benefits that appear only on specific platforms. There will be no loss for those who use the platforms not supported by your JNI features, and you will compete with native applications on equal terms where needed. The trick is to design an application that can take advantage of JNI features, but is still fully operational without them.

(Although Microsoft's virtual machine does not support JNI, it has many Windows-specific packages of features that give the same result: powerful native support. Unlike JNI extensions, however, you cannot replace these features with an alternate JNI library for an additional platform. If you use these Windows-specific features, you're committed to Windows and Microsoft's VM.)

### Bonus features

Start out with a 100% Pure Java application design. Think of the application as being contained entirely within the virtual machine, which is otherwise platform neutral. All your features must be based on whatever the Java platform has to offer. When you're finished, congratulate yourself! Your design will work on any Java Virtual Machine (JVM) running on any platform.

Next, look at your Java application from a user's point of view. It's probably not the only application running in his or her environment. What features is your application lacking that its native neighbors have? Run a few native applications together with yours, and ask yourself some questions about the differences:

* Do the native apps have special widgets that your application lacks? Tool tips are common, and so are context menus, but what about collapsing/expanding, minimizing/maximizing, and showing/hiding? Some GUIs have special popup messages and automatic behaviors. There's really no limit to what some people think users want.
* Do both types of app handle special mouse and key combinations in the same way? Java makes no special assumptions about the number of mouse buttons, but many platforms do. Keyboards are largely standardized, but there are exceptions; some keyboards (and some platforms) have more specialized keys than others. Native applications may exhibit special operating system behavior for special key combinations.
* Can your application interoperate with the natives? Can you drag and drop between them? What types of objects does each accept? Dragging files is common, but are other objects commonly dragged in or pasted?
* How does your application interact with the desktop? Some platforms have very cluttered desktops, supporting taskbars, flashing messages, context menus, and even Web content. Do the native applications use these features? (This article will handle the taskbar icon feature in Windows, with which you can add application-defined icons to the desktop taskbar that indicate status without requiring an open window.)
* How does your application interact with the operating system, as compared to the native applications? Do they handle special OS events, such as hibernation and input methods, in similar ways? Do they converse with system databases or registries? Do they write to a system event log?

Look at your list of missing features, and don't underestimate its importance. You may not like these features, finding them weird and difficult to understand, but that's just because you're not used to that platform. The average user knows them, uses them, and will expect to find them in your application. Unless your application provides functionality that is not available elsewhere, or is extremely cheap, users will always prefer native options.

But don't go overboard. Some features are so obscure or unpopular that their absence won't be noticed. Filter such features out, and list only the essentials. Try to think of these as features you want and not just obstacles to overcome. Thinking positively always helps.

The most important thing is to treat these features as an added benefit (or an annoying redundancy, if you prefer) for users of a specific platform, but never as features essential to the functioning of the application. Your app should be entirely and fully usable on either the Standard or Micro edition of the Java platform. This is the only way to protect the application's cross-platform portability -- and considering that portability was probably why you chose Java as a platform in the first place, it makes sense to protect it.

### Talking the talk

In our exercise above, you probably came up with a list of features your application lacked when compared to native applications running in the same environment. You may be surprised by how many you can implement purely in standard Java:

* The Java 2 platform has an improved Abstract Windowing Toolkit (AWT), and contains the Java Foundation Classes (JFC); the latter has a large set of lightweight, pure Java components capable of mimicking native platforms. The Pluggable Look and Feel (PLAF) architecture even lets you mimic an alternate look-and-feel architecture on any platform (within licensing limits) and customize your own. It is important to remember that the simulacrum is never the same as the original, and that there may be small quirks that the simulated PLAFs do not cover. As of JDK 1.2.2, Swing (JFC's GUI component set) is not very stable, but has a very flexible design and a bright future. JDK 1.3 (code-named Kestrel, and now available in beta) promises to be more stable and even more feature rich.
* JFC also supports the dragging and dropping of objects between JVMs, and between JVMs and native applications. Again, there are various quirks that need attention. Don't claim it works until you've tested it on every platform.
* The Java 2 platform also introduces standard extensions to the platform that may cover your needs. The Java Sound API and the Java 3D API are implemented with JNI. It is always best to use standard extensions, which are guaranteed to be widely supported on many platforms.

If you do decide to use JNI for your solution, don't immediately go native; keep the Java platform-neutral aloofness in mind. Consider each feature as if it were part of your virtual machine. Give it a name that is as generic as possible. Design the classes to be similar to those in the Java platform packages. Such aloofness is useful because there may be ways to simulate this feature on other platforms; when the time comes to do so, using platform-specific names and behavior models may seem out of place.

In this article, we will call our feature a desktop indicator, and will treat it with all the common Java courtesies, such as decorating it with listener classes. It's actually a fairly portable feature, and is available in some form or another on most desktop platforms.

### Designing the wrapper

We'll start by designing our Java wrapper from a non-native perspective. Under Win32, the taskbar icon works by sending messages to windows; this is necessary because of Win32's reliance on messaging to handle events. In Java, we prefer to use listener classes. Part of our implementation challenge will be to provide a native invisible window that receives the events and dispatches them to listeners. This design is much more flexible than one based on the Win32 model. In taking this route, we have overcome the temptation to send events to AWT windows, and enabled our feature to work without any AWT involvement.

Let's start with the listener:

public interface DesktopIndicatorListener

{

/\*\*

\* Called when a desktop indicator is clicked.

\*\*/

void onDesktopIndicatorClicked( DesktopIndicator source );

}

That's pretty straightforward.

As for our class, we will associate each instance with exactly one taskbar icon. We will need one static method to ensure that the library is loaded, and to indicate that the feature is supported on this VM:

/\*\*

\* Loads the JNI library, if available.

\*  
\*

\* Must be called before images are loaded and instances are created.

\*\*/

public static boolean initialize()

{

// Load JNI library

try

{

System.loadLibrary( "DesktopIndicator" );

}

catch( UnsatisfiedLinkError x )

{

return false;

}

return true;

}

Note that we are catching unsatisfied link exceptions because we want to deal gracefully with platforms that do not support this feature.

As for images, it makes sense to load them internally. Operating systems usually have internal structures for handling image lists that are far more efficient than loading them from a file. If we want to have a lot of blinking and icon changes, we will want the images loaded and cached before use. To make things simpler, we'll have one global image list for all instances, so this, too, will be a static method:

/\*\*

\* Loads an image file to memory.

\*  
\*

\* The image is loaded from the filename parameter, while the tool tip is

\* supplied in the tool tip parameter. The image file must be in a format

\* recognized by the native platform (for example, a ICO file for Win32).

\*  
\*

\* A return value of -1 indicates failure. Otherwise, it is the handle of

\* the image to be used for desktop indicators.

\*\*/

public static int loadImage( String filename )

{

try

{

return nativeLoadImage( filename );

}

catch( UnsatisfiedLinkError x )

{

return -1;

}

}

This method is a wrapper for a private native method that adds graceful handling of exceptions. A good convention for this kind of situation is to give the native version a native prefix, because otherwise these methods are identical:

private synchronized static native int nativeLoadImage( String filename ) throws UnsatisfiedLinkError;

Note that we are synchronizing the native method, rather than doing our own internal synchronization in the native code. The latter procedure is potentially tricky, since native synchronization services do not necessarily have anything to do with those of a JVM; a JVM implements its own threads. We'll go further into threads when we get into the native code.

The other methods are quite obvious. They handle the showing and hiding of the indicator, and the updating of the image and tool tip. We will add one special private field:

private int handler = 0;

This field is not used by the Java code at all, and is meant as a placeholder for the native code to keep a link to an equivalent underlying native class, called DesktopIndicatorHandler.

We now have our wrapper, and the design seems generic enough to be ported to other platforms. Let's go native!

### Walking the walk

Our native methods must somehow be converted to native code entry points. Javah, an integral tool of the JDK, generates a standard C/C++ header file from our Java source. Let's run Javah on our wrapper class:

> javah DesktopIndicator

Our C/C++ header starts with this:

/\* DO NOT EDIT THIS FILE - it is machine generated \*/

#include < jni.h >

/\* Header for class DesktopIndicator \*/

#ifndef \_Included\_DesktopIndicator

#define \_Included\_DesktopIndicator

#ifdef \_\_cplusplus

The header file includes jni.h, so it must be used in conjunction with the other C header files in the JDK that define various types and classes used.

The header file continues with prototypes of our entry point functions that are directly equivalent to the native Java methods:

extern "C" {

#endif

/\*

\* Class: DesktopIndicator

\* Method: nativeDisable

\* Signature: ()V

\*/

JNIEXPORT void JNICALL Java\_DesktopIndicator\_nativeDisable

(JNIEnv \*, jobject);

Although the functions assume C-calling conventions, the header files are C++-aware, and are easier to work with in C++, which, like Java, is an object-oriented member of the C family. We will use C++ for this example.

If required, you can develop your JNI library in languages other than C/C++, although you will have to do your own translation of entry points and arguments. In general, this is not recommended. You want your extension to be as small and independent as possible. Higher-level languages may require that various libraries be installed, which can complicate deployment considerably. Threading issues add even more difficulties.

Now that we have our prototypes to provide a skeleton, we need to fill in the meat. We'll be using Visual C++ 6.0 for this example, although any Win32 compiler with the system headers should work equally well. We will only be using Win32 API calls, and no features specific to Visual C++.

### The meat

We'll start by creating an empty Win32 project and workspace, making sure that it has the JDK's include folder in its inclusion path. Because we want our application to work on Windows 95, we will use the ANSI (as opposed to Unicode) versions of Win32 calls. Fortunately, JNI has the facilities needed to do the transformations.

(We're going to jump into some advanced JNI work here, so I won't cover the basics. If you find yourself getting lost, please refer to the [Resources](http://www.javaworld.com/jw-10-1999/jw-10-jni.html#resources) section below for more information on JNI.)

Each DesktopIndicatorHandler instance will need its own invisible window that will receive events and send them up to the Java instance, which, in turn, will delegate to its listeners.

We must consider carefully how to handle threading in our library, so let's look at the Win32 event model. Messages are not sent directly to windows, but put on an event queue owned by a thread. The thread must occasionally check for messages on its queue, and choose to either deal with them or delegate them to other callbacks. Each class of windows owned by the thread has its own callback that handles events according to the class behavior.

Although we synchronized these methods in Java, and can be sure that no more than one thread will call us before we leave our callbacks, we are still being called in arbitrary threads, meaning that each call may come from a different context. We cannot be sure that the invisible windows created in these contexts will get messages, because we cannot be sure that these threads (which are owned by the VM process) check their Win32 message queues and delegate messages. In this case, in fact, they don't. Because of this, we will create our own thread, with our own message queue handling, and make sure all our windows are created within its context. It is simple to asynchronously delegate work to our worker thread by posting custom messages to its message queue.

Creating our own native thread raises a problem, though. We will need to call Java methods from within our event-handling routine, and these will happen in the context of our own thread. The JVM, however, has its own multithreading scheme, and we cannot just intrude whenever we please. We need to have our own thread work politely with the JVM. Fortunately, JNI provides the facility for attaching our native threads to the VM and enabling them to work with the VM's synchronization scheme. When attaching the thread, we receive our own environment interface, through which we can make calls to the VM.

This is a delicate point, so we will illustrate it by including the full thread procedure:

DWORD WINAPI DesktopIndicatorThread::ThreadProc( LPVOID lpParameter )

{

DesktopIndicatorThread \*l\_this = (DesktopIndicatorThread \*) lpParameter;

// Attach the thread to the VM

l\_this->m\_vm->AttachCurrentThread( (void\*\*) &l\_this->m\_env, NULL );

MSG msg;

while( GetMessage( &msg, NULL, 0, 0 ) )

{

if( msg.message == WM\_DESKTOPINDICATOR )

{

// Extract handler

DesktopIndicatorHandler \*l\_handler = (DesktopIndicatorHandler\*) msg.lParam;

switch( msg.wParam )

{

case DesktopIndicatorHandler::enableCode:

l\_this->m\_handlerCount++;

l\_handler->doEnable();

break;

case DesktopIndicatorHandler::updateCode:

l\_handler->doUpdate();

break;

case DesktopIndicatorHandler::disableCode:

// Destroy it!

delete l\_handler;

// No more handlers?

if( !--l\_this->m\_handlerCount )

{

l\_this->m\_thread = 0;

// Detach thread from VM

l\_this->m\_vm->DetachCurrentThread();

// Time to die

ExitThread( 0 );

}

break;

}

}

else

{

TranslateMessage( &msg );

DispatchMessage( &msg );

}

}

// Detach thread from VM

l\_this->m\_vm->DetachCurrentThread();

return 0;

}

The thread begins by attaching itself to the JVM; the thread's environment pointer is stored in its instance. Note that this is a static function, so we are passing the this pointer by ourselves via the thread's private general-purpose parameter, which we sent in our CreateThread call.

Next, we will loop on our message queue. The GetMessage function blocks until a message arrives on the queue. We are using our own user-defined message code, with our custom identifier codes and a pointer to the relevant handler instance. The handler class does the real work, so we will simply delegate the operation to its methods in our Java-safe context.

The TranslateMessage and DispatchMessage calls will delegate messages to our invisible windows. This is important to note, because we are using our own user-defined messages, and must make sure that they have different codes; if we fail to take this precaution, our thread procedure will not be able to tell the difference.

To make the code cleaner, we will make the PostThreadMessage calls implicit. For example:

void DesktopIndicatorHandler::enable( JNIEnv \*env )

{

g\_DesktopIndicatorThread.MakeSureThreadIsUp( env );

while( !PostThreadMessage( g\_DesktopIndicatorThread, WM\_DESKTOPINDICATOR, enableCode, (LPARAM) this ) )

Sleep( 0 );

}

As seen in our thread procedure above, the message will cause doEnable to be called in the safe context. The weird loop and sleep setup is there because it may take a short while for the thread's message queue to initialize. Once the queue is up, PostThreadMessage should always return true. (Note that, although doEnable is private, DesktopIndicatorThread can call it because it is declared as a friend class. Yes, it's obvious, but it also may be a bit confusing with all this switching between Java and C++.)

The rest of the work is really straightforward Win32 programming. We register our window class:

// Register window class

WNDCLASSEX l\_Class;

l\_Class.cbSize = sizeof( l\_Class );

l\_Class.style = 0;

l\_Class.lpszClassName = TEXT( "DesktopIndicatorHandlerClass" );

l\_Class.lpfnWndProc = WndProc;

l\_Class.hbrBackground = NULL;

l\_Class.hCursor = NULL;

l\_Class.hIcon = NULL;

l\_Class.hIconSm = NULL;

l\_Class.lpszMenuName = NULL;

l\_Class.cbClsExtra = 0;

l\_Class.cbWndExtra = 0;

if( !RegisterClassEx( &l\_Class ) )

return;

We then create our invisible window (there's one for each handler instance):

// Create window

m\_window = CreateWindow

(

TEXT( "DesktopIndicatorHandlerClass" ),

TEXT( "DesktopIndicatorHandler" ),

WS\_POPUP,

0, 0, 0, 0,

NULL,

NULL,

0,

NULL

);

if( !m\_window )

return;

// Set this pointer

SetWindowLong( m\_window, GWL\_USERDATA, (LONG) this );

Note that we store our this pointer in the user area on the window. We do this in order to solve the same problem we had with the thread procedure -- our window procedure is also a static function.

We can now -- finally! -- create our taskbar icon:

// Add shell icon

NOTIFYICONDATA m\_iconData;

m\_iconData.cbSize = sizeof( m\_iconData );

m\_iconData.uFlags = NIF\_MESSAGE | NIF\_ICON | NIF\_TIP;

m\_iconData.uCallbackMessage = WM\_DESKTOPINDICATOR\_CLICK;

m\_iconData.uID = 0;

m\_iconData.hWnd = m\_window;

m\_iconData.hIcon = m\_icon;

strcpy( m\_iconData.szTip, m\_tooltip );

Shell\_NotifyIcon( NIM\_ADD, &m\_iconData );

We are using our own user-defined message for the callback, which is different from the message for the thread procedure, because they are both interpreted in the same place.

Our window procedure is simple:

LRESULT CALLBACK DesktopIndicatorHandler::WndProc( HWND hWnd, UINT uMessage, WPARAM wParam, LPARAM lParam )

{

// Check for our special notification message

if( ( uMessage == WM\_DESKTOPINDICATOR\_CLICK ) && ( INT(lParam) == WM\_LBUTTONDOWN ) )

{

DesktopIndicatorHandler \*l\_this = (DesktopIndicatorHandler \*) GetWindowLong( hWnd, GWL\_USERDATA );

// Click!

l\_this->fireClicked();

return 0;

}

else

return DefWindowProc( hWnd, uMessage, wParam, lParam );

}

It extracts the this pointer and fires a click. The fireClicked method delegates upward to the Java wrapper instance via JNI:

void DesktopIndicatorHandler::fireClicked()

{

g\_DesktopIndicatorThread.m\_env->CallVoidMethod( m\_object, m\_fireClicked );

}

Note that we are doing the call through the thread's environment, because that is the thread context in which we are running.

If you remember your JNI, you know that nonstatic native methods are called with a jobject parameter. But how did we match the jobject, which is a Java instance, with our C++ handler instance? Cast your mind back to that handler field in the wrapper ... I'm sure it's all coming back to you now, right? Here's the static C++ method to extract our pointer from the Java object:

DesktopIndicatorHandler \*DesktopIndicatorHandler::extract( JNIEnv \*env, jobject object )

{

// Get field ID

jfieldID l\_handlerId = env->GetFieldID( env->GetObjectClass( object ), "handler", "I" );

// Get field

DesktopIndicatorHandler \*l\_handler = (DesktopIndicatorHandler \*) env->GetIntField( object, l\_handlerId );

return l\_handler;

}

You may feel apprehensive about storing C++ pointers in Java instances. Don't worry; Java will not garbage-collect your C++ objects. This is, in fact, a standard technique for making objects opaque. As far as Java is concerned, the pointer is just a number stored in a field. In this case, Java makes no use of this number, and without knowing what it means, it's just data. In some cases, these numbers may be stored in Java and returned to native code by other native method calls. In such cases, the number is called a handle.

There's one last trick concerning this: we're storing the jobject in our handler instance, and using it to call the fireClicked method. This is dangerous, because it may be readily garbage-collected, in which case terrible things will happen if we reference it. It's easy to forget this, because we never worry about reference counting in Java. But we're not in Java, and must make sure that a reference is counted for us:

// Reference object

m\_object = env->NewGlobalRef( object );

We also must be careful to release the reference when it's no longer needed, or else the Java instance will never be garbage-collected:

// Release our reference

g\_DesktopIndicatorThread.m\_env->DeleteGlobalRef( m\_object );

The full source code is available online. See [Resources](http://www.javaworld.com/jw-10-1999/jw-10-jni.html#resources) for the URL.

Here is a small Java application to test the feature. Make sure the DLL and the ICO files are in your path:

public class DesktopIndicatorTest implements DesktopIndicatorListener

{

static DesktopIndicatorTest listener;

static int clicks = 4;

static int quickImage;

static int comicImage;

static public void main( String args[] )

{

// Initialize JNI extension

if( !DesktopIndicator.initialize() )

{

System.err.println( "Either you are not on Windows, or there is a problem with the DesktopIndicator library!" );

return;

}

// Load quick image

quickImage = DesktopIndicator.loadImage( "quick.ico" );

if( quickImage == -1 )

{

System.err.println( "Could not load the image file \"quick.ico\"!" );

return;

}

// Load comic image

comicImage = DesktopIndicator.loadImage( "comic.ico" );

if( comicImage == -1 )

{

System.err.println( "Could not load the image file \"comic.ico\"!" );

return;

}

// Create the indicator

DesktopIndicator indicator = new DesktopIndicator( quickImage, "Quick! Click me!" );

listener = new DesktopIndicatorTest();

indicator.addDesktopIndicatorListener( listener );

indicator.show();

// Instructions

System.err.println( "See the taskbar icon? Click it!" );

// Wait for the bitter end

try

{

synchronized( listener )

{

listener.wait();

}

}

catch( InterruptedException x )

{

}

// Time to die

indicator.removeDesktopIndicatorListener( listener );

indicator.hide();

System.err.println( "Goodbye!" );

}

public void onDesktopIndicatorClicked( DesktopIndicator source )

{

System.err.println( String.valueOf( clicks ) + " click(s) left..." );

// Countdown to death

clicks--;

if( clicks % 2 == 1 )

{

// Comic!

source.update( comicImage, "A message for you, sir!" );

}

else

{

// Quick!

source.update( quickImage, "Quick! Click me!" );

}

if( clicks == 0 )

{

// The end is nigh

synchronized( listener )

{

listener.notifyAll();

}

}

}

}

(This is a very bad application, because it does not offer the same functionality on other platforms. It does, however, fail rather gracefully.)

### Suggestions for improvement

The desktop indicator solution is certainly usable as is, but may be extended to offer even more functionality.

* Many native Windows applications feature a context menu for their taskbar icons. We can use Java to display this menu, but we must know where on the screen to display it. This information can be supplied in the listener's callback method.
* Currently, the desktop indicator only supports ICO files to render the icons; however, using JPEGs would be more interoperable with other Java packages, and it would be nice to use JPEG files located within JARs. It would not be trivial to support this feature; runtime translation of the desired JPEG to the ICO format used by Win32 would be necessary.

### GUI and beyond

JNI does have a serious limitation when it comes to extending Java's GUI: there is no standard way for getting native window handles from AWT components. Each VM may implement its AWT differently, and store the window handle in a different place. Future versions of the VM may handle windows in still different ways, ways that cannot be predicted with any certainty right now. With Java 2 platform VMs just starting to get a foothold (without this feature), I don't expect to see a standard way for doing this anytime soon.

This limitation means that it is quite impossible for your JNI extensions to alter or otherwise work with AWT-created components. You can definitely create your own mini-AWT framework to handle your platform-specific windowing features, but this requires considerable work, never integrates well, and is, of course, nonportable.

While I've focused this article on a GUI extension, there are other platform-specific features that users may expect that are not part of the GUI. One example is the Windows' registry. When running on Windows, Java applications cannot normally write their settings to the registry, and, as such, they will not be saved on the user's profile. A class that uses JNI to write to the registry on Windows, and writes to a file on other platforms, gives you the best of all worlds, without compromising its persistence features. I have supplied a rather primitive solution for this on my JNI site (see [Resources](http://www.javaworld.com/jw-10-1999/jw-10-jni.html#resources)).

### Support JNI ... carefully

I am firmly convinced that JNI extensions are not merely cool toys for Java, but are a real necessity if Java is to transcend the small world of applets and compete in the desktop-application arena. Java aficionados have hitherto downplayed JNI for an obvious reason: applications which rely on JNI features will no longer be portable, and Java will lose its main edge and become just another VM architecture. On the other hand, with careful design considerations, JNI extensions may save the day.

I am maintaining a small site for supporting public-domain open source JNI extensions similar to the one described in this article. If more people contribute, we can build a useful library of application boosters and make Java a viable platform for real-world, cross-platform applications. See [Resources](http://www.javaworld.com/jw-10-1999/jw-10-jni.html#resources) for the URL.

### The deployment issue

True native behavior is not the only problem with Java applications. Deployment is another big issue. Of course, you can deploy a Java application just like any other application, and there are several commercial kits for cross-platform installation, many running in Java. But it's still a far cry from deploying Java applets, and millions of miles away from HTML/DHTML Web applications, where all you have to do is turn on your Web server.

Wouldn't it be great if you could just give your users a URL for your applications and have them point their VMs to it? It would be just like using a browser, but with all the benefits of a real application with real windowing. The initiative is under way, and you are welcome to join and help where you can. It's all free software under the GNU General Public License. See [Resources](http://www.javaworld.com/jw-10-1999/jw-10-jni.html#resources) for more details.

This section presents the ReadFile example program. This example 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](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#examp)
  + [Native Method Declaration](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#native)
  + [Load the Library](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#load)
  + [Compile the Program](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#comp)
* [Generate the Header File](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#gen)
* [Method Signature](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#sig)
* [Implement the Native Method](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#impl)
* [Compile the Dynamic or Shared Object Library](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#sol)
* [Run the Example](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#run)

|  |
| --- |
|  |

**About the Example**

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

However, successfully running the program requires a few additional steps beyond compiling the Java 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 defintions 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[1](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/#TJVM) virtual machine. In this example, the loadFile function maps onto 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. You might need to configure your environment so the loadLibrary method can find your native code library.

static {

System.loadLibrary("nativelib");

}

**Compile the Program**

To compile the program, just 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 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

**Note:** When running javah on your own classes, be sure to use the fully-qualified class name.

**Method Signature**

The ReadFile.h header file defines the interface to map the Java language method to the native C function. It uses a method signature to map the arguments and return value of the Java language 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 hosuekeeping information.
* jobject: A reference to the method 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.

|  |
| --- |
| #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;  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);  }  jb=(\*env)->NewByteArray(env, finfo.st\_size);  (\*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 [Language Issues](http://java.sun.com/developer/onlineTraining/Programming/JDCBook/jniref.html#add) section shows how to map between Xbase database functions and Java language code
2. Use the shared stubs code available from the [JNI page](http://java.sun.com/products/jdk/faq/jnifaq.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 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  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 be able to find the native library. To do this, set the library path to the current directory as follows:

|  |
| --- |
| **Unix or Linux:**  LD\_LIBRARY\_PATH=`pwd`  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