JNIWrapper

Bringing worlds together

Tutorial

Version: *1.0*
Last Updated: *July 8, 2002*

Table of Contents

Introduction

Welcome to JNIWrapper for Microsoft Windows Tutorial. This tutorial is designed to give an example of creating a simple application with certain features not provided by the Java2 platform; it demonstrates most concepts that you need to know to develop successful applications using JNIWrapper.

The program we'll be writing is going to reveal the JNIWrapper abilities using the simple Win32 API functions. The first thing we'll try to do within its scope is to make some sounds using the system speaker. You don't usually hear much of that from a Java program, so it's the simplest way to make our application stand out. Therefore, we will call the program “Buzzer” according to the effect it produces. We will also add other nice features in the course of this guide.

The code will evolve as new concepts are introduced. By following the Tutorial step by step and adding the given code you will finally have a complete working program.

The whole Buzzer sample code is also available for download at Documentation page on JNIWrapper site.

Step 1: Setting Up

Environment Creation

First things first: let's set up the program working environment. Create a directory for the program, let's call it Sample: we'll store our Java file, JNIWrapper library and any other resources here. It will also be our working directory for the sample application. In the working directory let's create a directory for native files, calling it Bin. Now copy the Java library jniwrap.jar to the Sample directory, the native DLL (jniwrap.dll) and license (jniwrap.lic) files - to the Sample\Bin directory. Create the Java application file in the Sample directory; as has already been said, we'll call it Buzzer, basing on the effect it produces.

You should now have the following structure:

```
Sample\
       Bin\
           jniwrap.dll
           jniwrap.lic
       Buzzer.java
       jniwrap.jar
```

Creating Application

Now let's set up the class of the application, and run it before moving to the actual JNIWrapper coding.

Put the following code into Buzzer.java:

```
import com.jniwrapper.*;

public class Buzzer
{
    public Buzzer()
    {
    }

    public static void main(String[] args)
    {
        System.out.println("The Buzzer is running");
        Buzzer buzzer = new Buzzer();
    }
}
```

Compile the class:

```
javac -classpath jniwrap.jar Buzzer.java
```

and run the application:

```
java -classpath jniwrap.jar;. Buzzer
```

Throughout this tutorial under compiling and running the application we will understand performing two actions. Remember that you have to run the commands from the Sample directory for our program to work.

OK, we're all set – let's get to the real work.

Step 2: Working with Native Libraries

As we mentioned above, the first thing we'll try in our program is making sounds using the system speaker. To do this we'll use the standard Win32 function Beep. This function is exported from kernel32.dll, so our application requires this to be loaded. JNIWrapper provides a shortcut method for calling a function that loads a library and invokes its method. But the library itself provides lots of interesting functions, so we'll better load it once to get access to the complete variety of functions that we may choose to invoke.

Preparing Search Path

Before doing any library loading we need to prepare a search path for our libraries, namely JNIWrapper DLL and those we will use for invoking functions. By default, libraries are located by the `DefaultLibraryLoader` singleton instance. This class searches for libraries following the path specified by the `java.library.path` system property, though, using its addPath method, we can add new directories there. All standard Win32 libraries are on the system PATH which is automatically added to `java.library.path` so we just have to add a search path for the JNIWrapper DLL. (We could have copied it to some location on the system PATH, but it is generally considered to be not a good style, especially for deployed applications.) The native code DLL is in the Bin folder and since we run only from the current directory let's use the relative path to it. We'll also use the relative paths later to keep the example simpler. So, in our main function we add a line as shown below:

```
DefaultLibraryLoader.getInstance().addPath("Bin");
```

You will need to configure the library loader in all programs that use JNIWrapper before any native-related activity takes place.

Loading Native Code Libraries

Let's get back to our library. To load a library you should create an object of the type `com.jniwrapper.Library` specifying the library name. Since kernel32.dll will be used often in our program we'll create a field for it and load it in the constructor. Now we change our class as follows (the new code is given in bold):

```
private Library _kernel;

public Buzzer()
{
    _kernel = new Library("kernel32");
}
```

There is no need to specify the extension – it is appended automatically. Libraries are loaded when their objects are constructed. You don't have to worry about freeing libraries: if a library object is no longer used – the native library defined by it is unloaded. Now let's compile and run our application to make sure that everything is typed in correctly and all libraries can be found.

Step 3: Using Simple Types

In this chapter we'll do actual beeping by invoking a native function with simple parameters. Let's put our beeping code in the new method called buzz.

As mentioned above, we'll use the Beep function to produce sounds. This function takes two DWORD parameters and returns a BOOL. We are not really interested in the returned value, since this function is highly unlikely to fail, so we'll ignore the returned value. We have to pass values, however, so we create objects for function parameters:

```
private void buzz()
{
    // Prepare beep parameters: low and high frequencies and beep durations
        UInt32 low = new UInt32(400);
        UInt32 high = new UInt32(1000);
        UInt32 durLow = new UInt32(200);
        UInt32 durHigh = new UInt32(200);
```

We use standard JNIWrapper type `UInt32` which is exactly what `DWORD` is: a 32-bit unsigned integer. Our method will produce a series of lower and higher frequency beeps, that's why we define two sets of parameters: [`low`; `durLow`] and [`high`; `durHigh`]. In JNIWrapper all Numeric types have constructors that specify initial value and also have getters and setters for the value. All parameters in JNIWrapper are mutable – they behave like variables. We could have used one pair of parameters and modify their values between calls, but it's less expressive so let's have two sets.

Step 4: Invoking Functions

Now, that we have the library and parameters only one thing is missing – the function to invoke. Functions are obtained from the library that exports them. To get the function you should use the name it's exported with. Luckily most Win32 API functions are exported with the same (or almost the same – we'll get to that case later) names with which they appear in the documentation. To get our Beep function we use the following code:

```
            // Obtain function reference from the library
            Function beep = _kernel.getFunction("Beep");
```

Everything is ready to do the buzz. We create a small loop in which the Beep function is invoked to make high and low frequency beeps five times.

```
            // Do the beeping
            for (int i = 0; i < 5; i++)
            {
                beep.invoke(null, high, durHigh);
                beep.invoke(null, low, durLow);
            }
```

Notice that the first argument of the invoke method is `null`. This indicates that we do not care for the return value. Later we can easily add checking for it. Imagine the consequences of incorrectly ignoring the return value in the conventional JNI implementation: to get it later you would have to modify the native method signature, change the native implementation and Java usages and rebuild both Java and native code. This is only one example of how JNIWrapper can save you a lot of development time.

Our new method is complete – let's invoke it from the main method to see how it works:

```
        public static void main(String[] args)
        {
            // Add JNIWrap.dll directory to library loader search path
            DefaultLibraryLoader.getInstance().addPath("Bin");
            Buzzer buzzer = new Buzzer();
            buzzer.buzz();
        }
```

Compile and run the application – you should hear an alarm-like sound from the speaker.

Step 5: Using Strings

The next goal for our application will be to allow user to run only one instance of it at a time. Many Java programs try to achieve this by binding a particular socket. This approach does not ensure success because you can not be absolutely sure the selected socket number is good.

Our program will use a different method of creating a named mutex on start and checking for its existence to find out if another instance is running. With a well chosen name this method is practically bullet-proof. To create a mutex we need to specify its name which is a string.

String Types

In Java we have only one type of strings: the Unicode ones. Native side, however, has both single-byte (char*) and wide (wchar_t*) strings. JNIWrapper supports both types providing two concrete parameter classes: `AnsiString` for single-byte strings and `WideString` for wide strings. All strings we create in this application will consist only of 7-bit pure ASCII characters, so we'll use AnsiStrings.

To make sure that our single-instance mechanism is working properly we'll need to make our program stay running for as long as needed. One of the simplest ways to achieve this is to display a message box at the place where we would like to wait. Of course, we can use JOptionPane here, but its not in the spirit of this tutorial and it does not produce that cool native notification sound either. Let's prepare a method that would allow displaying information or error message box:

```
        private void showMessageBox(String message, int flags)
```

The first argument here is the message itself while the second is the flags mask for the `MessageBox` function we are going to invoke. Let's take a closer look at the `MessageBox` function. From the documentation we see that it has the following signature:

```
int MessageBox(
  HWND hWnd,          // handle to owner window
  LPCTSTR lpText,     // text in message box
  LPCTSTR lpCaption,  // message box title
  UINT uType          // message box style
);
```

Creating Mutex and Displaying Message

We have to create a correct parameter set to invoke this function. The simplest first: the type `UINT` has a directly corresponding type in JNIWrapper – `UInt`. Now, `HWND`: it's a handle and, therefore, a pointer. In general, pointers are more complex than the things we would like to touch in this chapter. Luckily, the data this pointer is pointing to will be never used, we do not have to fill any data in this pointer's referenced area and the last, but not least, in our case this pointer is simply `NULL`. On our platform we could use a UInt32 parameter since we know that all pointers are unsigned 32-bit integers, but a more correct way would be to use a Pointer.Void parameter. We'll set its handle value to zero. This corresponds to a C-like conversion from an integer. Whenever you need a constant such as (HWND)-1 use Pointer.Void specifying the value in the constructor or setter.

The rest of the parameters are strings. The type LPCTSTR defines a pointer to a string that is ANSI or Unicode, depending on the build configuration. Under the cover, it means that there are two versions for this function for each of the string formats. In Win32 API their names are created from a function name by appending A for the ANSI version or W for Unicode. We would like to use ANSI, so our target function is spelled MessageBoxA. The arguments are string pointers, so to pass them we create instances of AnsiString. Strings in C/C++ are pointers to string data. In JNIWrapper string parameters actually represent string data which, to become string arguments, have to be pointed to; but when we are calling a function, its string parameters are auto-magically passed to the underlying function as a pointer to relieve programmers from the burden of creating any extra pointers. Therefore, the string parameters for the MessageBoxA function are defined as just AnsiString.

Here's the argument preparation code for our method:

```
        Pointer.Void hWnd = new Pointer.Void(0);
        AnsiString text = new AnsiString(message);
        AnsiString caption = new AnsiString("Buzzer");
        UInt uFlags = new UInt(flags);
```

To invoke the MessageBoxA function we'll need another library – user32.dll. Add the following code (in bold) to the declarations and constructor:

```
    private Library _kernel;
    private Library _user;

    public Buzzer()
    {
        _kernel = new Library("kernel32");
        _user = new Library("user32");
    }
```

Let's finish the showMessage method:

```
        Function messageBox = _user.getFunction("MessageBoxA");
        messageBox.invoke(null, hWnd, text, caption, uFlags);
```

Here we also can safely ignore the return value.

For further convenience we'll create methods to display the error and information message boxes; the methods will specify the required style constants for those message box types.

```
    private void message(String message)
    {
        showMessageBox(message, 0x30);
    }

    private void error(String message)
    {
        showMessageBox(message, 0x10);
    }
```

With what we have already learned it is really easy to write the mutex-based single instance checking part.

First create a parameter representing a mutex name:

```
        private boolean checkOneInstance()
        {
            AnsiString mutexName = new
    AnsiString("com.jniwrapper.sample.BuzzerMutex");
```

Then try to open a mutex with this name:

```
            UInt32 desiredAcces = new UInt32(0x1F0001);
            Bool inheritHandle = new Bool(false);
            Pointer.Void mutexHandle = new Pointer.Void();
            Function openMutex = _kernel.getFunction("OpenMutexA");
            openMutex.invoke(mutexHandle, desiredAcces, inheritHandle, mutexName);
```

Here we are interested in the result: if there is no such mutex, the function returns NULL, otherwise we will know that another instance is running and we'll have to close the new instance. To get the return value we need to create a variable of the required type and pass it as the first argument to the invoke method. After the invocation completes the passed parameter contains the returned value. Next, we test the returned value:

```
            if (!mutexHandle.isNull())
            {
                // Mutex exists - one instance is already running
                return false;
            }
```

Now if the mutex doesn't exist this instance is the first one and we would like to create a locking mutex. Creating is not much different from checking except for the part where the NULL result now indicates the error we would like to catch. Here is the rest of the checking method:

```
            // Not yet running - lock by creating mutex
            Pointer.Void mutexAttributes = new Pointer.Void(0);
            Bool initialOwner = new Bool(false);
            Function createMutex = _kernel.getFunction("CreateMutexA");
            createMutex.invoke(mutexHandle, mutexAttributes, initialOwner,
    mutexName);
            if (mutexHandle.isNull())
            {
                throw new RuntimeException("Mutex creation failed, last error = "
    + getLastError(true));
            }
            return true;
        }
```

You have probably noticed that the last piece of the code references a method that is not yet defined: getLastError. It has to return the last system error code, optionally clearing the error status depending on a boolean parameter. Let's implement it now. We already know all methods and types to write the following simple piece of code:

```
        private long getLastError(boolean clear)
        {
            UInt32 r = new UInt32();
            Function getLE = _kernel.getFunction("GetLastError");
            getLE.invoke(r);
            long errCode = r.getValue();
            if (clear)
            {
                UInt32 zero = new UInt32(0);
                Function setLE = _kernel.getFunction("SetLastError");
                setLE.invoke(null, zero);
            }
            return errCode;
        }
```

We are ready to finish this iteration by adding the following (bold) code to the main method:

```
    public static void main(String[] args)
    {
        // Add JNIWrap.dll directory to library loader search path
        DefaultLibraryLoader.getInstance().addPath("Bin");
        Buzzer buzzer = new Buzzer();
        if (!buzzer.checkOneInstance())
        {
            buzzer.error("Buzzer is already running");
            System.exit(0);
        }
        buzzer.message("Buzzer started, click OK to close");
    }
```

Compile and run the program: you should see the “Buzzer started...” message box – don't close it yet, try to run another instance. Now you should see an error box saying that our application is already running.

Step 6: Using Callbacks

In this iteration let's create a timer to make a signal after some time runs out. We'll be using Windows native timers here, so we would need a way to have the native code to call the Java code. JNIWrapper provides such capability through the `com.jniwrapper.Callback` class. You can create any number of callbacks that can have any parameters and return values.

Creating a callback is simple – you have to subclass the `Callback` class, define the callback arguments and returned value, and implement the callback method (`callback`). Then you pass an instance of this class as a parameter wherever you need a reference to the callback.

Creating Timer Callback

We are going to use the Windows API function `SetTimer`. It accepts the reference to the `TimerProc` callback function, so we need to implement such a callback. Here's its definition:

```
VOID CALLBACK TimerProc(
  HWND hwnd,         // handle to window
  UINT uMsg,         // WM_TIMER message
  UINT_PTR idEvent,  // timer identifier
  DWORD dwTime       // current system time
);
```

After examining the function signature we can see that we know how to pass any of such types into a function call. Specifying them as callback arguments is no harder. Let's define our callback class as an inner class of our application class so that we can easily call the useful methods defined there:

```
    private class TimeOutCallback extends Callback
    {
        private Pointer.Void _hwnd = new Pointer.Void();
        private UInt _msg = new UInt();
        private UInt _timerID = new UInt();
        private UInt32 _time = new UInt32();

        public TimeOutCallback()
        {
```

We have defined the fields for each of our callback parameters. There is no return value so we do not define any parameter for it. In the constructor we have to configure our callback by specifying parameters of the callback signature:

```
            init(new Parameter[] {
                _hwnd,
                _msg,
                _timerID,
                _time
            }, null);
        }
```

The last `null` is the void return value. Now just to make this class complete and compilable we'll implement the callback method. We will make it just buzz for now and will add more intelligent code to it later in this iteration:

```
            public void callback()
            {
                buzz();
            }
        }
```

The callback created is already usable. Easy, isn't it? Just imagine doing all this stuff using the conventional JNI techniques!

Using Callback in the Application

Now we need to create a timer and pass the callback reference to be called when the timer elapses. To do this, we'll create a method and call it startTimer. This function requires passing a window handle. We don't have that one yet, so to get a handle quickly now we'll use the sun.awt.windows.WToolkit class from the Java implementation. In subsequent iterations we will get our own window handle without messing with internal implementation classes. For now let's create a Java window and use its handle. Add the following code (in bold) to the class initialization:

```
import com.jniwrapper.*;

import java.awt.*;

import sun.awt.windows.WToolkit;

import javax.swing.*;

public class Buzzer
{
    private Library _kernel;
    private Library _user;

    private Window _window;
    private static final int TIMER_ID = 1;

    public Buzzer()
    {
        _kernel = new Library("kernel32");
        _user = new Library("user32");
        _window = new JWindow();
        _window.setVisible(true);
    }
```

Notice the new constant – it will be used in the code below. The window will just provide its handle to hook the timer to.

Let's implement the startTimer method:

```
    private void startTimer(long timeout)
    {
        int h =
WToolkit.getWToolkit().getNativeWindowHandleFromComponent(_window);
        Pointer.Void hWnd = new Pointer.Void(h);
        UInt eventID = new UInt(TIMER_ID);
        UInt timeOutVal = new UInt(timeout);
```

```
            UInt result = new UInt();
            TimeOutCallback timeOutCallback = new TimeOutCallback();
            Function setTimer = _user.getFunction("SetTimer");
            setTimer.invoke(result, hWnd, eventID, timeOutVal, timeOutCallback);
```

Passing a callback is even more straightforward than its implementation: just create an object and pass it to the function that requires it. It's just like an event listener. Some sanity checking and we're done:

```
            if (result.getValue() == 0)
            {
                throw new RuntimeException("Failed to create a timer, error code =
    " + getLastError(true));
            }
        }
```

Just before we move on, let's implement the stopTimer method to be called from the callback:

```
        private void stopTimer(Pointer.Void hwnd, UInt timerID)
        {
            Function killTimer = _user.getFunction("KillTimer");
            killTimer.invoke(null, hwnd, timerID);
        }
```

Here we used JNIWrapper types as parameters, because we already have them in the callback and there is no need to unwrap the values just to wrap them back.

Let's test the resulting code. Modify the main method as follows:

```
        public static void main(String[] args)
        {
            // Add JNIWrap.dll directory to library loader search path
            DefaultLibraryLoader.getInstance().addPath("Bin");
            Buzzer buzzer = new Buzzer();
            if (!buzzer.checkOneInstance())
            {
                buzzer.error("Buzzer is already running");
                System.exit(0);
            }
            buzzer.startTimer(5000);
        }
```

Compile the program. If it is already running there is no control to terminate it: to close the program just press ctrl-C in its console (remember not to use javaw.exe here or you'll have to use the Task Manager to stop the buzzing). Run the program. You should hear beeping regularly in 5 second intervals until the program is terminated. This means that the timer we are using is repetitive so we will have to stop it when it is no longer needed.

Now we'll complete the callback code with a few useful lines. We would like to (a)stop timer, (b)notify the user that the timer has elapsed, and (c)finish the program. Here's the whole implementation of the callback method that does it all:

```
            public void callback()
            {
                stopTimer(_hwnd, _timerID);
                buzz();
                message("Timer has elapsed!");
                System.exit(0);
            }
```

Notice that though our main thread terminates before the timer elapses the program still lives, because of non-daemon AWT threads that keep our window alive.

Take a look at the first line of the function. When the callback is invoked the variables specified at its creation are set to the argument values before the `callback` method is called. We pass two of the arguments to the `stopTimer` function. Our callback does not have a return value, if it had, we would have to assign the return value parameter before leaving the `callback` method. There are no limitations of what a callback can do other than those specified for the original callback.

The iteration is over. Compile and run the program. Five seconds after being started it will emit the usual buzzing and then pop up the information box saying that the timer has elapsed. Closing this message box will terminate the program.

Step 7: Using Structures

In the previous iteration we have used an implementation-specific class to get the window handle. In this iteration we are going to get one in a more legitimate way. We will create a native window and later make it custom shape like a splash screen. Creating and managing a window using Windows API requires passing more complex parameters than we have used before, namely structures and pointers. We will begin with structures. Any window has its event queue and requires some thread to dispatch its events. Messages are placed into the message queue in form of the `MSG` structures. Let's examine the contents of that structure and implement it using JNIWrapper. Here is its definition:

```
typedef struct tagMSG {
  HWND   hwnd;
  UINT   message;
  WPARAM wParam;
  LPARAM lParam;
  DWORD  time;
  POINT  pt;
} MSG, *PMSG;
```

One of the `MSG` structure members is a structure itself: `POINT pt`. Its layout is as follows:

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

This one is really simple – it looks like a good starting point to learn how to create structures in JNIWrapper. Structures are implemented by creating an instance of the `com.jniwrapper.Structure` class. Most of them, however, are often reused and so it is usually better (and more readable) to create a subclass for each structure required. We'll choose the second approach:

```
    private static class POINT extends Structure
    {
        public LongInt _x = new LongInt();
        public LongInt _y = new LongInt();
```

First we create a variable for each structure member. These are to hold actual member data and to be used for accessing it. Next, the structure must be initialized defining its layout. Members are passed to the `init` method in the order they appear in the native-side structure declaration. Optionally, the structure alignment can be specified, but it is not needed here since all Windows API structures have the default alignment (of 1). Here is the rest of the code for this structure:

```
        public POINT()
        {
            init(new Parameter[] {_x, _y});
        }
    }
```

The `POINT` structure is ready. Using the same principle, let's create a class for the MSG structure:

```
private static class MSG extends Structure
{
    Pointer.Void _hwnd = new Pointer.Void();
    UInt _message = new UInt();
    UInt32 _wParam = new UInt32();
    UInt32 _lParam = new UInt32();
    UInt32  _time = new UInt32();
    POINT _pt = new POINT();

    public MSG()
    {
        init(new Parameter[]{_hwnd, _message, _wParam, _lParam,
                             _time, _pt});
    }
}
```

Notice that defining a member of a complex type (structure) is no different from defining any of simple ones. Uniformity is one of the JNIWrapper main design goals.

Step 8: Using Pointers

To demonstrate the use of a pointer in JNIWrapper let's implement the event loop for our program. We'll do this *before* creating a window and doing all other preparation stuff, because using a pointer here is most simple and straightforward. In fact, the only missing part will be the window handle, so let's assume that it is already stored in the `_hSplash` variable defined as follows:

```
        private Pointer.Void _hSplash;
```

Add the above line to the declaration section of the class file.

Creating Window Message Loop

We'll implement the message loop in the run method of our program. The loop consists of sequential invocation of three API functions: GetMessage, TranslateMessage and DispatchMessage. However, instead of taking the MSG structure as their argument they all require LPMSG, i.e. a pointer to that structure. In JNIWrapper pointers are similar to any other type: to create a pointer you have to create an instance of the com.jniwrapper.Pointer class. Each pointer should point to some other parameter called a referenced object. Whenever a pointer is written or read its referenced object is also written or read respectively. Let's create a pointer to the `MSG` structure:

```
        private void run()
        {
            Function getMessage = _user.getFunction("GetMessageA", null);
            Function translateMessage = _user.getFunction("TranslateMessage",
    null);
            Function dispatchMessage = _user.getFunction("DispatchMessageA",
    null);
            MSG msg = new MSG();
            Pointer msgPointer = new Pointer(msg);
```

The code related to the pointer creation is marked bold. In this example `msgPointer` is a pointer and `msg` is its referenced object. In the previous code we used the `Pointer.Void` class instead of pointers, because we did not care for the referenced object and did not have to provide one. In this case we have to provide the `MSG` structure so we have to use the real pointer. The rest of this function code uses the created pointer just as any other parameter:

```
            Bool result = new Bool();
            for (;;)
            {
                getMessage.invoke(result, msgPointer, _hSplash, new UInt32(0), new
    UInt32(0));
                if (!result.getValue())
                {
                    break;
                }
                translateMessage.invoke(null, msgPointer);
                dispatchMessage.invoke(null, msgPointer);
            }
        }
```

Although there is no need for us to access or modify data in the msg structure, we could easily do so.

Pointers and Strings

Now that we have learned all the techniques we can go into the gory details of creating a window. First we will define the most complex structure in this example: the equivalent of Windows API WNDCLASS structure. We have already seen that any type can become a structure member equally easily. This structure will be another example – it will hold a callback reference. Add this class to your code:

```
        private static class WndClass extends Structure
        {
            private UInt32 _style = new UInt32();
            private Callback _lpfnWndProc;
            private Int32 _cbClsExtra = new Int32();
            private Int32 _cbWndExtra = new Int32();
            private Pointer.Void _hInstance = new Pointer.Void();
            private Pointer.Void _hIcon = new Pointer.Void();
            private Pointer.Void _hCursor = new Pointer.Void();
            private Pointer.Void _hbrBackground = new Pointer.Void();
            private AnsiString _lpszClassName = new AnsiString();

            public WndClass(Callback windowProc, String className, Pointer.Void
    bgBrushHandle)
            {
                _style.setValue(3); // CS_HREDRAW | CS_VREDRAW
                _lpfnWndProc = windowProc;
                _lpszClassName.setValue(className);
                _cbClsExtra.setValue(0);
                _cbWndExtra.setValue(0);
                _hInstance.setValue(0);
                _hIcon.setValue(0);
                _hCursor.setValue(0);
                _hbrBackground.setValue(bgBrushHandle.getValue());
                init(new Parameter[] {_style, _lpfnWndProc, _cbClsExtra,
                                      _cbWndExtra, _hInstance, _hIcon, _hCursor,
                                      _hbrBackground, new Pointer.Void(0),
                                      new Pointer(_lpszClassName)});
            }
        }
```

Take a look at the bold code. When defining structures that contain strings you should always remember that there can be two types of them: pointers to character data and character arrays. To distinguish between the two look at the member definition. Ones defined as char *name; are pointers and those defined as char name[20]; are arrays. In function calls strings are always passed as pointers so string parameters are automatically converted, but in structures there is no way to know, so the programmer should explicitly specify how the string is stored. If it is a pointer, a Pointer instance must be passed as the corresponding structure member. If it is an array, you should create a string with the maximum length equal to that of the expected character array and pass that parameter itself as the structure member. In this case we have a pointer to characters version.

Finishing Up

Now we are ready to finish the application. To do this we have to implement a window procedure callback, register a window class, create a window, shape and center it, and define a window painting method. There is nothing special in either of these tasks so you can just take a look at the final code provided along with this tutorial. After running it you should see a small round window (staying on top of all others, by the way), in ten seconds the computer should emit a beeping sound and pop up a message box saying that the timer has elapsed. Of course, the single instance rule is still enforced.

The tutorial is over. By now you know everything you need to start successfully using JNIWrapper in your applications.