Chapter 1. Introduction to J/Invoke

This chapter introduces J/Invoke, and walks through its features using a series of small examples.

We begin with setting up our development environment and downloading and installing J/Invoke. We then wet our feet with a quick example that calls the Win32 API from Java. The next section discusses J/Invoke features and introduces the helper classes in the com.jinvoke.win32 package that make calling the Win32 API even easier. This is followed by a tour of J/Invoke annotations, enumerations, and classes, interspersed with several small examples that highlight important features. The Data Type Conversions section spells out the general rules for converting between native types and Java types, and provides a table that can help you when writing J/Invoke declarations for Win32 functions.

Java does not have pointers, but native functions often use them. J/Invoke lets you utilize single-element Java arrays to simulate pointers. Some native functions require pointers to callback functions that they can call later. We will see how J/Invoke can let Java methods be treated as callback functions by the Win32 API. Finally, we discuss how to deploy your Java applications that use J/Invoke.

Setting up your development environment

J/Invoke requires a Java Development Kit (JDK) that supports annotations (i.e. JDK 5 or higher). The latest JDK can be downloaded from http://java.sun.com/javase/downloads/index.jsp.

This chapter assumes you have installed JDK in the default installation directory, and added the path to the JDK\bin folder to your System path. To do this,

1. Right click on My Computer and click Properties.
2. In the System Properties dialog, open the Advanced tab.
3. In the Advanced tab, click on the Environment Variables button.
4. In the Environment Variables window that opens up, select the Path variable in System variables section, and click the Edit button.
5. Append the path to your JDK\bin folder (such as “C:\Program Files\Java\jdk1.6.0_03\bin”) to the variable value separated from the existing values by a semicolon(;), and
6. Click OK

This will add java.exe and javac.exe to the system path, so they can be called directly from the command line (cmd.exe), without specifying the full path.

Downloading and Installing J/Invoke

Download J/Invoke from http://www.jinvoke.com/download, and extract the contents of the zip file.

As shown in Figure 1-1, this will create a folder in the C:\ drive called **jinvoke** with the following contents:

- **jinvoke.jar** - the J/Invoke runtime library
- **doc folder** - contains the J/Invoke Developer Guide and API Reference (javadoc)
- **samples folder**- contains many J/Invoke samples and tutorials referred to in the documentation

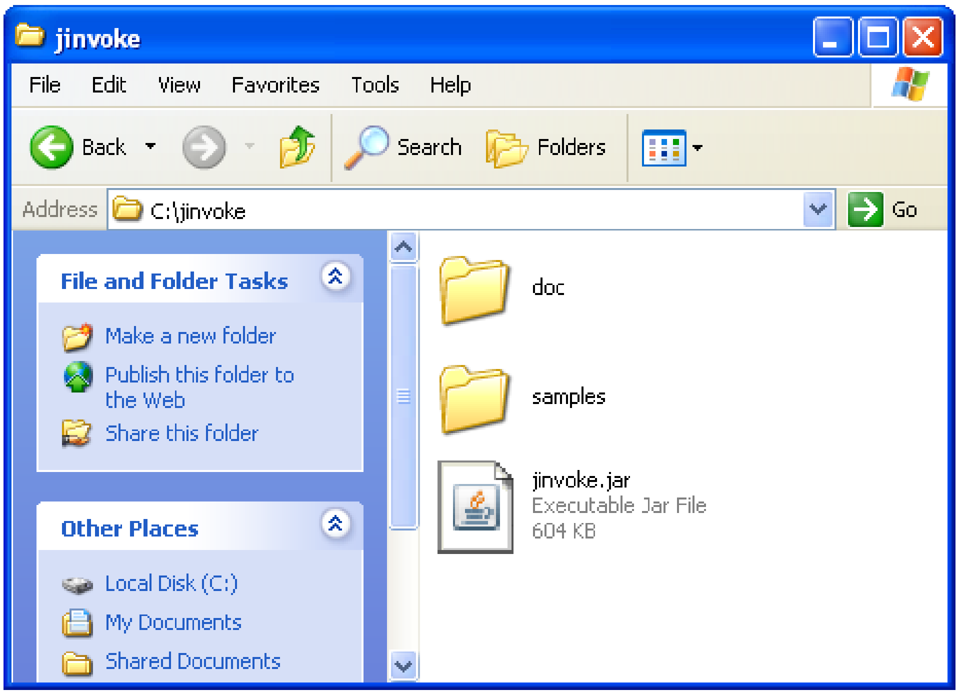


Figure 1-1 Contents of the J/Invoke folder

The **jinvoke.jar** file is the only runtime component needed by J/Invoke. It is a self contained JAR file that includes the native libraries needed by J/Invoke at runtime. When a program using J/Invoke is run for the first time, it extracts a helper DLL called jinvoke.dll to the same folder as jinvoke.jar. This DLL is used by J/Invoke internally at runtime. There is no need to configure the java library path or distribute the DLL separately.

If you are using an IDE, add **jinvoke.jar** to your project's build path/classpath. If using the command line to compile or run your Java code, you will need to add jinvoke.jar to the classpath, as we will see shortly.

A Welcoming Example

Now that you have J/Invoke installed, let's call the Win32 API from a small Java program.

We'll utilize the **`MessageBox()`** function from the Win32 API. This function, documented at http://msdn2.microsoft.com/en-us/library/ms645505(VS.85).aspx, displays a message box with a title, message, and an optional icon. We could have used the JOptionPane class provided by Swing for this, but here we want to learn how to call the Win32 API from Java.

So, let's dive right in. Fire up your favorite text editor or IDE and create a file called HelloWindows.java with the following contents:

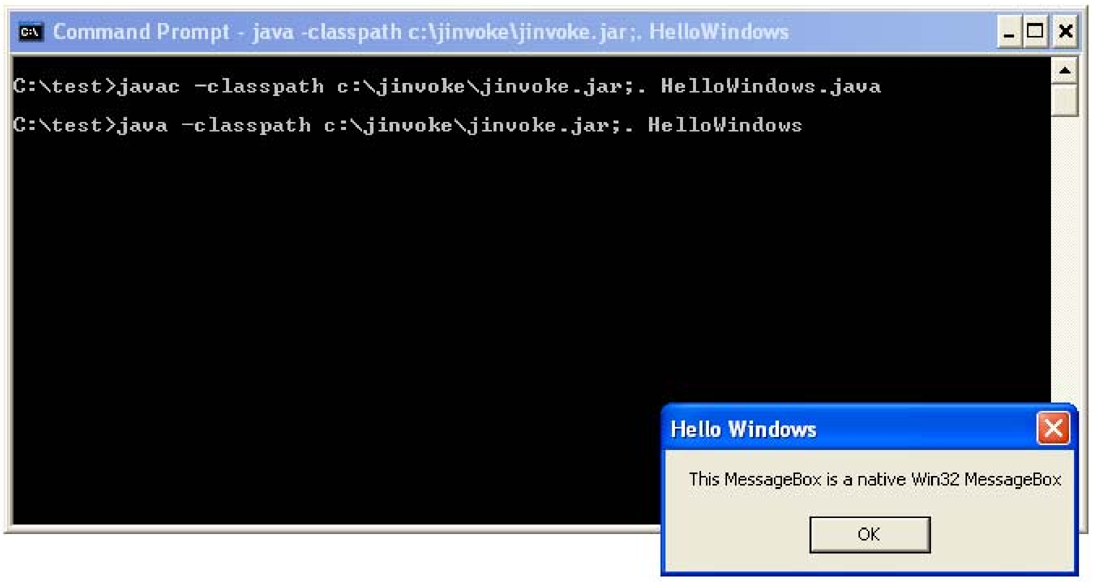
```
import com.jinvoke.JInvoke;
import com.jinvoke.NativeImport;

public class HelloWindows {

   @NativeImport(library="User32")
   public static native int MessageBox(int hwnd,
                                       String text,
                                       String caption,
                                       int type);

   public static void main(String[] args) {
      JInvoke.initialize();
      MessageBox(0, "This MessageBox is a native Win32 MessageBox",
                    "Hello Windows", 0);
   }
}
```

This example, together with all the book's code, can be downloaded from http://www.jinvoke.com/win32hacks.

Save HelloWindows.java in a new folder, C:\test. To compile and run this program, open a command window, and type the following commands:

```
cd C:\test
javac –classpath C:\jinvoke\jinvoke.jar;. HelloWindows.java
java –classpath C:\jinvoke\jinvoke.jar;. HelloWindows
```

If all goes well, you should see a Win32 MessageBox pop-up, as shown in Figure 1-2.



```
C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. HelloWindows.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. HelloWindows
```

Command Prompt - java -classpath c:\jinvoke\jinvoke.jar;. HelloWindows
Hello Windows
This MessageBox is a native Win32 MessageBox
OK

Figure 1-2 Compiling and running the test program

Let's go over the code to see how it works. First, we declare a native method with the @NativeMethod annotation, specifying its Win32 DLL. In this case, User32.dll exports the MessageBox() function. This method signature matches the Win32 function prototype described at http://msdn2.microsoft.com/en-us/library/ms645505(VS.85).aspx.

```
@NativeImport(library="User32")
public static native int MessageBox(int hwnd,
                                    String text,
                                    String caption,
                                    int type);
```

Next, we initialize the J/Invoke runtime using `JInvoke.initialize()`. This statement tells the J/Invoke runtime to load the native DLL, and link the native methods declared in this class with the specified exported DLL functions.

Finally, we invoke the method. That's all that's needed to invoke the Win32 API using J/Invoke.

If Murphy's Law kicks in, you might get the following error:

```
Exception in thread "main" java.lang.UnsatisfiedLinkError:
Example.MessageBox(IL
java/lang/String;Ljava/lang/String;I)I
        at Example.MessageBox(Native Method)
        at Example.main(Example.java:10)
```

This can happen if you forget to call `JInvoke.initialize()`. In the absence of this method, Java assumes that you will provide a JNI DLL to implement the native method for MessageBox(). By itself, Java doesn't know about J/Invoke and ignores the @NativeImport annotation. Java expects to find a JNI DLL containing the implementation of this native method, and because there is none, you get the error.

Calling `Jinvoke.initialize()` causes the native methods in that class to be linked to the same-named methods from the specified Win32 DLL, i.e. from User32.dll in this example.

But you might also get the following error:

Exception in thread "main" java.lang.UnsatisfiedLinkError: Unable to resolve Msg Box or MsgBoxW in User32
    at com.jinvoke.JInvoke.InitMethod(Native Method)
    at com.jinvoke.JInvoke.initialize(JInvoke.java:145)
    at Example.main(Example.java:9)

This error indicates that the DLL doesn't export a function of the specified name. This could be because you've named the method wrongly, such as MsgBox() instead of MessageBox().

If you want to keep the Java method name different from the Win32 function name, you can do it by specifying the Win32 name explicitly in the @NativeImport annotation, as shown below:

```
@NativeImport(library="User32", function="MessageBox")
public static native int showMessage(int hwnd,
                                     String text,
                                     String caption,
                                     int type);
```

Calling the Win32 API

To make calling the Win32 API easier, the **com.jinvoke.win32** package provides helper classes that contain J/Invoke function declarations for the most commonly used Win32 DLLs. Import the class you need, and directly call the Java method corresponding to the Win32 API you want.

The available APIs are declared in a set of classes, one for each Windows DLL, as shown in the following table.

J/Invoke Win32 Helper Class	**DLL Name**	**Functionality**
com.jinvoke.win32.Kernel32	Kernel32.dll	Base services
com.jinvoke.win32.Gdi32	Gdi32.dll	Graphics device interface
com.jinvoke.win32.User32	User32.dll	User interface

com.jinvoke.win32.Advapi32	Advapi32.dll	Crypto API, event logging
com.jinvoke.win32.Shell32	Shell32.dll	Windows shell API
com.jinvoke.win32.Winmm	Winmm.dll	Multimedia
com.jinvoke.win32.WinInet	WinInet.dll	Internet

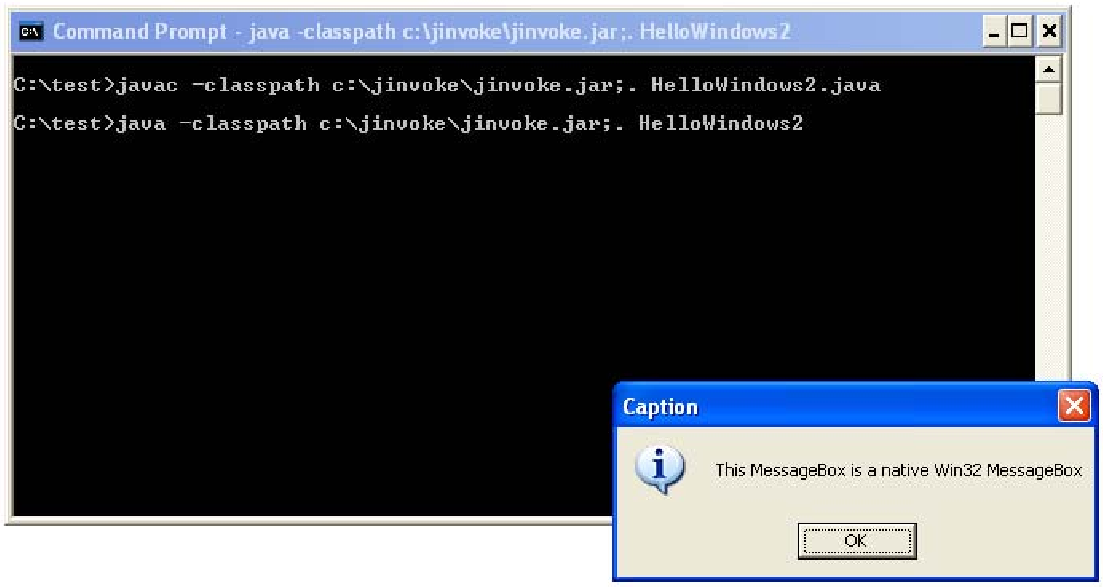
The Win32 API uses many constants and structures which are defined in **`com.jinvoke.win32.WinConstants`** class and the **`com.jinvoke.win32.structs`** package respectively.

For instance, **`MessageBox()`** is contained in User32.dll and therefore declared in the User32 class. Import this class and you can call `MessageBox()` directly. Thus, the previous example could be simplified to:

```
import com.jinvoke.win32.User32;
import static com.jinvoke.win32.WinConstants.*;

public class HelloWindows2 {
   public static void main(String[] args) {
      User32.MessageBox(0,
                        "This MessageBox is a native Win32 MessageBox",
                        "Hello Windows", MB_ICONINFORMATION|MB_OK);
      }
}
```

Figure 1-3 shows the output of this program. It adds an 'information icon' to the message box, because we use the *`MB_ICONINFORMATION`* type in conjunction with *`MB_OK`*, both of which are documented at http://msdn2.microsoft.com/en-us/library/ms645505(VS.85).aspx and defined for convenience in `com.jinvoke.win32.WinConstants`.

When using the Win32 helper classes, you do not need to call JInvoke.initialize(). It has already been called in their static initializers.



```
Command Prompt - java -classpath c:\jinvoke\jinvoke.jar;. HelloWindows2

C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. HelloWindows2.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. HelloWindows2
```

Caption

This MessageBox is a native Win32 MessageBox

Figure 1-3 Calling the Win32 API using helper classes in com.jinvoke.win32 package

The J/Invoke API

J/Invoke has a simple and easy-to-understand API: there are just three annotations, three classes and a couple of enumerations. We’ll start by briefly overviewing each one, and then illustrate their use in several small examples.

First, the annotations:

- The **@NativeImport annotation** - this provides the information needed to call a function exported from a native DLL. We met this annotation in the first example earlier in the chapter.
- The **@NativeStruct annotation** - this annotation is used to represent a native structure (a C struct) as a Java class.
- The **@Embedded annotation** - this annotation is employed to define fixed-size strings and arrays in structs.

Next, the classes:

- The **JInvoke class** has just one static method – initialize(). This method must be called once in each class before any native methods marked with the @NativeImport annotation are called. We saw initialize() in the first example of this chapter.
- The **Callback class** converts a Java method into a callback function that can be called by native code.
- The **Util class** provides numerous utility methods. These include ways to convert Java types to native types and vice versa, methods for obtaining the size of structs,

pointers, and chars. GUI applications can make use of Util methods to convert Windows icons into Java icons, and to obtain window handles for Java windows.

Finally, there are a couple of enumerations:

- The **CallingConvention enumeration** is used to specify the calling convention for a native method. By default, J/Invoke assumes a method employs the Win32 API default calling convention (Stdcall), so this enumeration isn't usually needed.
- The **Charset enumeration** indicates how a Java String is converted when passed to native code. This is used by the @NativeImport and @NativeStruct annotaions.

That was a quick tour of the complete J/Invoke API. *No, really!*

The NativeImport annotation

The @NativeImport provides J/Invoke with the information required to link to a native DLL, and call an exported function. The annotation's parameters specify the library name, the name of the exported function, its character set (i.e. if it accepts Unicode or Ansi strings), and a calling convention. It's basic form was seen in the examples earlier in this chapter.

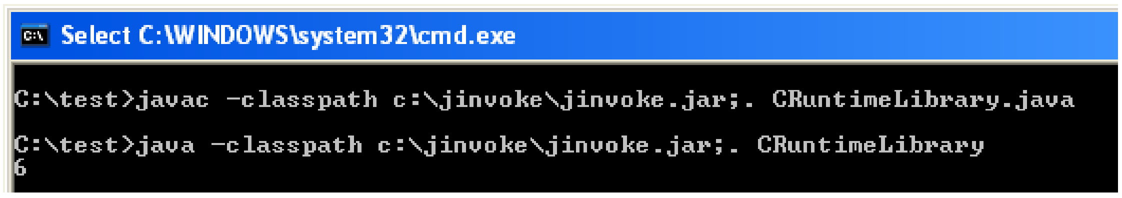
The **charset** element of the @NativeImport annotation is useful when the DLL function accepts Ansi string parameters. By default, J/Invoke converts Java Strings to Unicode (wide character) strings when calling functions, as shown in the earlier MessageBox() examples. This works well for most of the Win32 API, but there are some functions, like those in the C runtime library, that work with Ansi (char) strings. To call such functions, specify the **charset** element as Ansi.

The **convention** element of the @NativeImport annotation specifies the calling convention for the DLL function. For most of the Win32 API, the default Stdcall calling convention is a safe bet. However, the C Runtime library (in msvcrt.dll) uses the Cdecl calling convention.

The following example calls strlen() from the C Runtime Library documented at http://msdn2.microsoft.com/en-us/library/78zh94ax(VS.80).aspx. As a consequence, the Ansi charset and the Cdecl calling convention are employed:

```
import com.jinvoke.*;

public class CRuntimeLibrary {
    @NativeImport(library="msvcrt",
              charset=Charset.ANSI,
              convention=CallingConvention.CDECL)
    public static native int strlen(String str);

    public static void main(String[] args) {
        JInvoke.initialize();
        System.out.println(strlen("abcdef"));
    }
}
```

This program prints out the number of characters in "abcdef", as shown in Figure 1-4.



Select C:\WINDOWS\system32\cmd.exe

```
C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. CRuntimeLibrary.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. CRuntimeLibrary
6
```

Figure 1-4 Calling strlen(), a C-runtime library function

The data types of the native Java method arguments and return value should correspond to the types of the DLL function parameters and return type. See the **Data type conversions** section below to see how Java types map to native types.

The NativeStruct annotation

The @NativeStruct annotation applied to a Java class indicates that the class represents a native structure (a C struct).

The Java class should be declared as public, be in it's own .java file, and contain public fields corresponding to the members of the native struct, declared in the same order. A struct may contain fields of primitive types, strings (including StringBuffer and StringBuilder), callbacks (converted to function pointers), nested structs, embedded arrays, embedded strings, and arrays of primitives and structs.

For example, the C declaration of the SYSTEMTIME struct defined at http://msdn2.microsoft.com/en-us/library/ms724950(VS.85).aspx is as follows:

```
typedef struct _SYSTEMTIME {
    WORD wYear;
    WORD wMonth;
    WORD wDayOfWeek;
    WORD wDay;
    WORD wHour;
    WORD wMinute;
    WORD wSecond;
    WORD wMilliseconds;
} SYSTEMTIME, *PSYSTEMTIME;
```

This struct can be represented in Java as:

```
import com.jinvoke.NativeStruct;

@NativeStruct
```

```
public class SYSTEMTIME {
    public short wYear;
    public short wMonth;
    public short wDayOfWeek;
    public short wDay;
    public short wHour;
    public short wMinute;
    public short wSecond;
    public short wMilliseconds;
}
```

The GetSystemTime() function, defined in MSDN at http://msdn2.microsoft.com/en-us/library/ms724390.aspx, retrieves the current system date and time. The C declaration of the function is:

```
void WINAPI GetSystemTime(
   __out  LPSYSTEMTIME lpSystemTime
);
```

The function is passed a pointer to an empty SYSTEMTIME struct which it fills with values.

In Java, the function is declared as:

```
@NativeImport(library="kernel32")
static native void GetSystemTime(SYSTEMTIME pst);
```

Internally, J/Invoke converts the struct into a byte array, and passes a pointer to that array into the native function. The function uses the pointer to write into the struct members. On returning from the function call, J/Invoke reads back the updated byte array, and updates the members of the SYSTEMTIME object.

The following example uses the SYSTEMTIME struct and GetSystemTime() function.

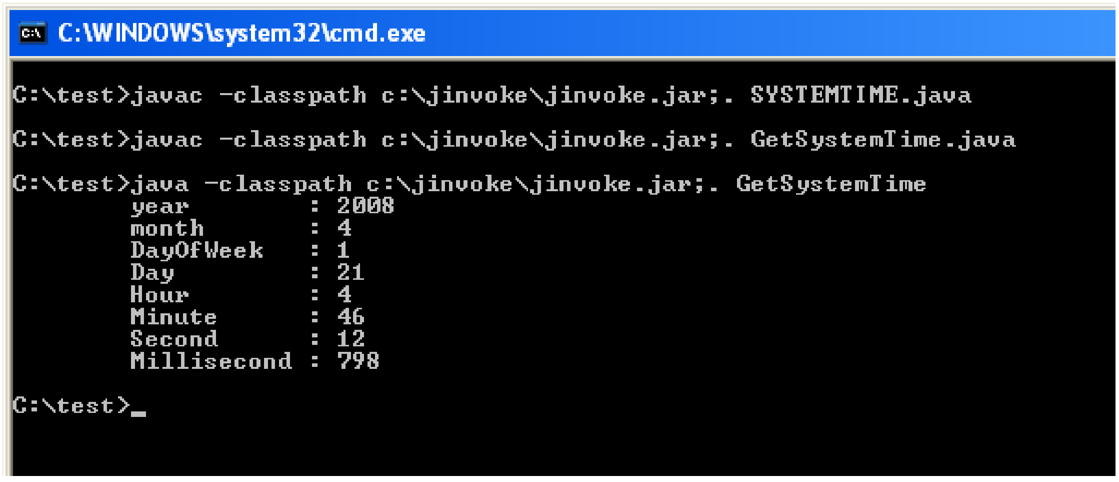
```
import com.jinvoke.JInvoke;
import com.jinvoke.NativeImport;

public class GetSystemTime {

   @NativeImport(library="kernel32")
   static native void GetSystemTime(SYSTEMTIME pst);

   public static void main(String[] args) {
      JInvoke.initialize();
      SYSTEMTIME systemtime = new SYSTEMTIME();
      GetSystemTime(systemtime);

      System.out.println(
         "\tyear        : " + systemtime.wYear +
         "\n\tmonth       : " + systemtime.wMonth +
         "\n\tDayOfWeek   : " + systemtime.wDayOfWeek +
         "\n\tDay         : " + systemtime.wDay +
         "\n\tHour        : " + systemtime.wHour+

         "\n\tMinute      : " + systemtime.wMinute+
         "\n\tSecond      : " + systemtime.wSecond+
         "\n\tMillisecond : " + systemtime.wMilliseconds);
    }
}
```

Note that we first initialize the struct using ‘new’:

```
SYSTEMTIME systemtime = new SYSTEMTIME();
```

If we do not initialize the struct, a null pointer will be passed into GetSystemTime(), resulting in an access violation.

The output of the program is shown in Figure 1-5.



```
C:\WINDOWS\system32\cmd.exe

C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. SYSTEMTIME.java

C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. GetSystemTime.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. GetSystemTime
        year        : 2008
        month       : 4
        DayOfWeek   : 1
        Day         : 21
        Hour        : 4
        Minute      : 46
        Second      : 12
        Millisecond : 798

C:\test>_
```

Figure 1-5 Using the SYSTEMTIME struct and GetSystemTime() function

The Embedded annotation

The @Embedded annotation is used to declare either fixed-size strings or arrays embedded in structures. We’ll consider each kind of embedding in turn.

Fixed size Strings embedded in Structures

In Hack 2.1, we will encounter the SHFILEINFO struct, that is used to obtain shell related file information. It is defined at http://msdn2.microsoft.com/en-us/library/aa453689.aspx:

```
typedef struct _SHFILEINFO {
  HICON hIcon;
  int iIcon;
  DWORD dwAttributes;
  TCHAR szDisplayName[MAX_PATH];
  TCHAR szTypeName[80];
} SHFILEINFO;
```

This struct can be represented by a Java class with the @NativeStruct annotation. The `szDisplayName` member is an embedded String of length MAX_PATH, which is defined as 260 characters at http://msdn2.microsoft.com/en-us/library/aa365247.aspx. The `szTypeName` member is an embedded String of length 80 characters. The corresponding Java declaration is:

```
package com.jinvoke.win32.structs;
import com.jinvoke.Embedded;
import com.jinvoke.NativeStruct;

@NativeStruct
public class ShFileInfo {
    public int hIcon;
    public int iIcon;
    public int dwAttributes;

    @Embedded(length=260)
    public StringBuffer szDisplayName = new StringBuffer(260);

    @Embedded(length=80)
    public StringBuffer szTypeName = new StringBuffer(80);
}
```

We need to use the @Embedded annotation for embedded String members, set its length member, and initialize their values to StringBuffers of the declared capacity.

Fixed size Arrays embedded in Structures

Consider the GUID structure defined at http://msdn2.microsoft.com/en-us/library/aa373931(VS.85).aspx:

```
typedef struct _GUID {
  DWORD Data1;
  WORD Data2;
  WORD Data3;
  BYTE Data4[8];
} GUID;
```

This struct can be represented by a Java class with the @NativeStruct annotation. J/Invoke maps int to DWORD and short to WORD. The Data4[] member is an embedded byte array of 8 bytes, and is expressed using the @Embedded annotation as shown below:

```
import com.jinvoke.Embedded;
import com.jinvoke.NativeStruct;

@NativeStruct
public class GUID {
    public int Data1;
    public short Data2;
    public short Data3;

    @Embedded(length=8)
    public byte[] Data4 = new byte[8];
}
```

We’ll encounter the GUID struct in Chapter 6 when we use a USB human interface device (HID) to provide input to a Java program.

Data Type Conversions

J/Invoke automatically manages the conversion of Java types to native C types. The native type of each parameter and return type is inferred from the signature of the declared native method in Java.

For example, in the earlier MessageBox() examples, J/Invoke automatically converts the String arguments from Java Strings to null terminated C strings, and the Java int arguments to native 32 bit integers.

J/Invoke follows a small set of conversion rules:

- Java primitives are converted into their respective native types.

- Strings are converted to null terminated C strings (char or wchar_t based on the charset). If Strings can be modified by the native function, StringBuffer or StringBuilder should be utilized. This is because Java Strings are immutable - they cannot be modified once constructed.

- Classes with @NativeStruct annotations are converted into native structures. The previous section contains examples of this rule.

- Arrays are converted into pointers to arrays of the corresponding native type. An example follows in the next section.

- Java methods can be wrapped in Callback objects to represent function pointers for callbacks from native code. There’s a short example in the Callback section later in this chapter.

The following table gives a summary of the type conversions used by J/Invoke. A more exhaustive table is included with the J/Invoke documentation.

Java type	**Corresponding native types**	**Native representation**	**Notes/Comments**
byte	BYTE, INT8, SBYTE	8 bit integer	Can be used as method arguments and return types. For passing unsigned values, use the signed types as a two's complement representation.
short	short, short int, SHORT, INT16, WORD	16 bit integer	
int	int, long, long int, int32, INT, UINT, LONG, DWORD, LPARAM, WPARAM,…	32 bit integer	
long	__int64, INT64, LONGLONG	64 bit integer	
float	float, FLOAT	32 bit (single precision) floating point number	
double	Double, DOUBLE, long double	64 bit (double precision) floating point number	
char	char, CHAR *when charset=Ansi, or* wchar_t, WCHAR *when charset=Unicode*	8 bit char (Ansi) 16 bit widechar (Unicode)	
boolean	BOOL	32 bit integer	
Boolean	VARIANT_BOOL	16 bit integer	
String	const char* string *when charset=Ansi, or* const wchar_t * string *when charset=Unicode*	Pointer to null terminated const string	Used when constant Strings are to be passed to DLL functions. Can be used as return value.
StringBuffer StringBuilder	char* string *when charset=Ansi, or* wchar_t * string *when charset=Unicode*	Pointer to null terminated mutable (non-const) string. Set the capacity large enough to hold the largest string	Not allowed as return type, but value can be modified by the called DLL function. Set the capacity large

		that the DLL function can generate.	enough to hold the largest String the DLL function can return.
Class annotated with @NativeStruct	Pointer to struct	Pointer to native struct	Can be used for method arguments and return types.
Callback	Callback Function	Function pointer	Parameter type only.
Arrays	Pointer to array of respective primitive type or struct	Pointer to array	Parameter type only. Can be modified by the DLL function.
void	VOID		Return type only.

A few things to keep in mind :

- The long type is only 32 bits in C/C++, not 64 bit as in Java, so a Java int should be employed when the native parameter is of type long.

- There is no direct representation of unsigned integer types in Java – so, use the signed types as a two's complement representation.

Simulating Pointers using Single Element Arrays

Java does not have pointers, but they can be simulated using arrays, where the array has only one element.

Some DLL functions return a value by having the caller pass in a pointer to the value to be updated. The called function updates the value by dereferencing the pointer. To utilize such a function in Java, simply pass it a single element array. On its return, the first element of that array will contain the value set by the called function.

One such function is GetDiskFreeSpace(), declared at http://msdn2.microsoft.com/en-us/library/aa364935(VS.85).aspx:

```
BOOL WINAPI GetDiskFreeSpace(
  __in    LPCTSTR lpRootPathName,
  __out   LPDWORD lpSectorsPerCluster,
  __out   LPDWORD lpBytesPerSector,
  __out   LPDWORD lpNumberOfFreeClusters,
  __out   LPDWORD lpTotalNumberOfClusters
);
```

`lpRootPathName` is a null terminated native string, and the remaining arguments are pointers to DWORDs (of type LPDWORD or 'long pointer to DWORD'). The function is passed the root directory of a disk (say C:\ or D:\) and returns the amount of free space available on the disk along with other disk related information such as the cluster size. The __in and __out attributes on the function parameters indicate the directionality of the arguments. The caller passes in the first argument, and the function returns the other arguments using call-by-reference.

This function is called in C by passing it pointers to DWORDs, as shown below:

```
// C code example
DWORD sectorsPerCluster, bytesPerCluster, numberOfFreeClusters,
      totalNumberOfClusters;

GetDiskFreeSpace(rootname, &sectorsPerCluster, &bytesPerCluster,
                           &numberOfFreeClusters, &totalNumberOfClusters);
```

On the function's return, the DWORD parameters contain values.

The corresponding Java signature uses arrays as arguments wherever the native function uses pointers, as shown below:

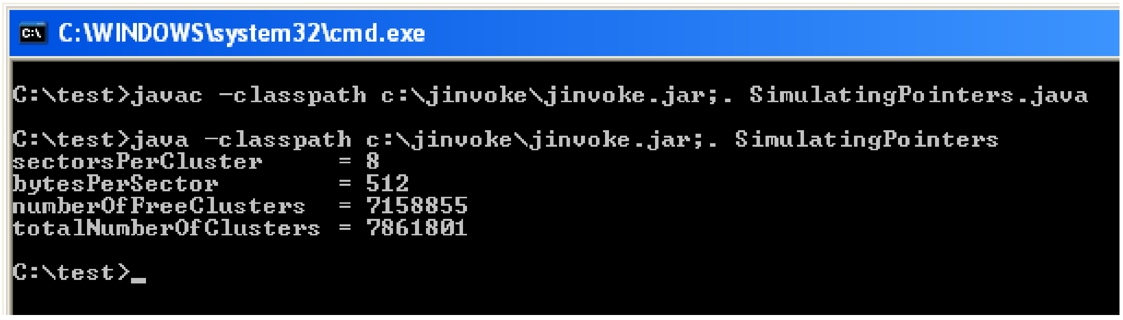
```
   @NativeImport(library="Kernel32")
   public static native boolean GetDiskFreeSpace(
            String lpRootPathName,
            int[] lpSectorsPerCluster,
            int[] lpBytesPerSector,
            int[] lpNumberOfFreeClusters,
            int[] lpTotalNumberOfClusters);
```

This method is called in Java by passing it single-element arrays for the call-by-reference parameters. On the method's return, the first element of each array holds a value.

Sample code follows:

```
import com.jinvoke.JInvoke;
import com.jinvoke.NativeImport;

public class SimulatingPointers {

   @NativeImport(library="Kernel32")
   public static native boolean GetDiskFreeSpace(String lpRootPathName,
            int[] lpSectorsPerCluster,
            int[] lpBytesPerSector,
            int[] lpNumberOfFreeClusters,
            int[] lpTotalNumberOfClusters);

   public static void main(String[] args) {
      JInvoke.initialize();

      // create single element arrays to pass into the function
      int[] sectorsPerCluster     = { 0 };     // create a one element array,
      int[] bytesPerSector        = { 0 };     // initialized with int 0

      int[] numberOfFreeClusters  = new int[1];// using the alternate single
      int[] totalNumberOfClusters = new int[1];// element array creation syntax


      GetDiskFreeSpace("C:\\", sectorsPerCluster, bytesPerSector,
                               numberOfFreeClusters, totalNumberOfClusters);

      // query the first element of the arrays...
      // these have been populated by the native function
      System.out.println("sectorsPerCluster     = "+ sectorsPerCluster[0]);
      System.out.println("bytesPerSector        = "+ bytesPerSector[0]);
      System.out.println("numberOfFreeClusters  = "+ numberOfFreeClusters[0]);
      System.out.println("totalNumberOfClusters = "+ totalNumberOfClusters[0]);
   }
}
```

Figure 1-6 shows the result of running this program.



```
C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. SimulatingPointers.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. SimulatingPointers
sectorsPerCluster     = 8
bytesPerSector        = 512
numberOfFreeClusters  = 7158855
totalNumberOfClusters = 7861801

C:\test>_
```

Figure 1-6 Using single element arrays to simulate pointers

The com.jinvoke.Util class provides additional methods for working with pointers, including casting them to structs, strings, and byte arrays, and vice versa. We'll see many examples in later chapters.

Callbacks - "Don't contact me, I'll contact you"

Some Win32 API functions require a pointer to a callback function as a parameter. The native code calls the function in response to some internal event.

In Java, a method is not a first class object – in other words, a method cannot be directly employed as a function argument. To enable a Java method to be passed as a callback function, J/Invoke provides the `com.jinvoke.Callback` class. An instance of this class represents a Java method, and can be passed to a native function when a function pointer is expected. When the native code calls the callback function, J/Invoke intercepts the call and invokes the specified Java method.

Any public Java method of any class can be converted into a Callback object by using one of Callback's constructors.

Let’s look at an example. The EnumWindows() function, documented at http://msdn2.microsoft.com/en-us/library/ms633497(VS.85).aspx, repeatedly calls a callback function, passing it the handle for each top-level window it finds.

The C declaration of the EnumWindows function is the following:

```
BOOL EnumWindows(
    WNDENUMPROC lpEnumFunc,
    LPARAM lParam
);
```

`lpEnumFunc` is a pointer to an application-defined callback function. We’ll substitute a Callback object for that pointer in our Java code.

The LPARAM argument is a 32-bit long, which EnumWindows() uses as a pointer to a string.

The corresponding function using J/Invoke is:

```
@NativeImport(library = "user32")
public static native boolean EnumWindows(Callback callPtr, String lPar);
```

The signature of the lpEnumFunc() callback function is defined as EnumWindowsProc() at http://msdn2.microsoft.com/en-us/library/ms633498(VS.85).aspx:

```
BOOL CALLBACK EnumWindowsProc(
    HWND hwnd,  // the window handle
    LPARAM lParam
);
```

We need to provide a function that matches this signature:

```
public static boolean EnumWindowsProc(int hwnd, String lParam) {
   // function body here...
   // this function is called once for each top-level window
}
```

We must convert our EnumWindowsProc() method into a Callback object and pass it to the EnumWindows() method:

```
Callback callback = new Callback(EnumerateWindows.class, "EnumWindowsProc");
EnumWindows(callback, "data");
```

The first argument of the Callback() constructer is the class where EnumWindowsProc() is defined (i.e. inside the EnumerateWindows class).

EnumWindows() calls the EnumWindowsProc() method for each top-level window, and supplies the window’s handle as an argument. The method can use the handle to access the window’s title:

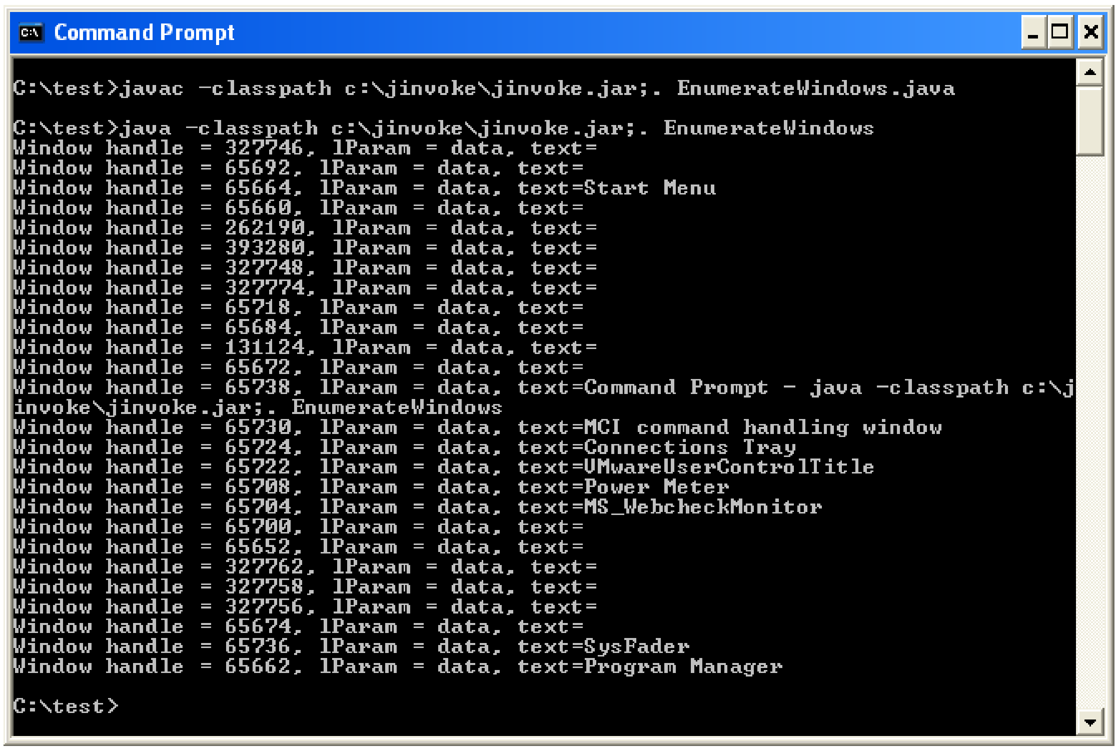
```
public static boolean EnumWindowsProc(int hwnd, String lParam) {
   StringBuffer sb = new StringBuffer(128);
   GetWindowText(hwnd, sb, sb.capacity());
   System.out.println("Window handle = " + hwnd + ", lParam = " + lParam +
                                                   ", text=" + sb);
   return true;
}
```

GetWindowText() comes from User32.dll, and is defined at http://msdn2.microsoft.com/en-us/library/ms633520.aspx. It isn't part of the User32 helper class in the version of J/Invoke that we're using, so must be defined with a @NativeImport annotation.

The following class brings everything together, demonstrating how EnumWindows() uses a callback function in Java.

```
import com.jinvoke.Callback;
import com.jinvoke.JInvoke;
import com.jinvoke.NativeImport;

public class EnumerateWindows {

   @NativeImport(library = "user32")
   public static native int EnumWindows(Callback callPtr, String lPar);

   @NativeImport(library = "user32")
   public static native int GetWindowText(int hWnd, StringBuffer sb,
                                                    int nMaxCount);

   public static void main(String[] args) {
      JInvoke.initialize();

      Callback callback = new Callback(EnumerateWindows.class,
                                "EnumWindowsProc");
      EnumWindows(callback, "data");
   }

   public static boolean EnumWindowsProc(int hwnd, String lParam) {
      StringBuffer sb = new StringBuffer(128);
      GetWindowText(hwnd, sb, sb.capacity());
      System.out.println("Window handle = " + hwnd + ", lParam = " + lParam +
                                                   ", text=" + sb);
      return true;
   }

}
```

Figure 1-7 shows the output of running this program. It prints out the Window handle, the lParam argument, and the window title that it obtains using GetWindowText().



```
C:\test>javac -classpath c:\jinvoke\jinvoke.jar;. EnumerateWindows.java

C:\test>java -classpath c:\jinvoke\jinvoke.jar;. EnumerateWindows
Window handle = 327746, lParam = data, text=
Window handle = 65692, lParam = data, text=
Window handle = 65664, lParam = data, text=Start Menu
Window handle = 65660, lParam = data, text=
Window handle = 262190, lParam = data, text=
Window handle = 393280, lParam = data, text=
Window handle = 327748, lParam = data, text=
Window handle = 327774, lParam = data, text=
Window handle = 65718, lParam = data, text=
Window handle = 65684, lParam = data, text=
Window handle = 131124, lParam = data, text=
Window handle = 65672, lParam = data, text=
Window handle = 65738, lParam = data, text=Command Prompt - java -classpath c:\j
invoke\jinvoke.jar;. EnumerateWindows
Window handle = 65730, lParam = data, text=MCI command handling window
Window handle = 65724, lParam = data, text=Connections Tray
Window handle = 65722, lParam = data, text=VMwareUserControlTitle
Window handle = 65708, lParam = data, text=Power Meter
Window handle = 65704, lParam = data, text=MS_WebcheckMonitor
Window handle = 65700, lParam = data, text=
Window handle = 65652, lParam = data, text=
Window handle = 327762, lParam = data, text=
Window handle = 327758, lParam = data, text=
Window handle = 327756, lParam = data, text=
Window handle = 65674, lParam = data, text=
Window handle = 65736, lParam = data, text=SysFader
Window handle = 65662, lParam = data, text=Program Manager

C:\test>
```

Figure 1-7 Enumerating all top-level windows using a callback function

Figure 1-7 lists several windows with no title. Most of there are invisible system windows.

Callbacks are not tied to static methods in a class - they can also be used with instance methods of objects, as shown in the code below:

```
import com.jinvoke.Callback;
import com.jinvoke.JInvoke;
import com.jinvoke.NativeImport;

public class EnumWindowsCallback {

   @NativeImport(library="user32")
   public static native int EnumWindows(Callback callPtr, String lPar);

   @NativeImport(library="user32")
   public static native int GetWindowText(int hWnd, StringBuffer sb,
                                                    int nMaxCount);

   public static void main(String[] args) {
      JInvoke.initialize();
      EnumWindowsCallback test2 = new EnumWindowsCallback();
      test2.enumwindows();
   }

   private void enumwindows() {
      EnumWindows(new Callback(this, "EnumWindowsProc"), "data");
   }
   public boolean EnumWindowsProc(int hwnd, String lParam) {
      StringBuffer sb = new StringBuffer(128);
      GetWindowText(hwnd, sb, sb.capacity());
      System.out.println("Window handle = " + hwnd + ", lParam = " + lParam +
                                                    ", text=" + sb);
      return true;
   }
}
```

The output of this program is identical to the one in Figure 1-7.

Deployment

Since J/Invoke uses standard Java and JNI internally, it can be deployed using all the standard Java deployment techniques. You can utilize J/Invoke with desktop applications, Java Web Start applications, applets, JSPs and servlets, web services, and Rich Client Platform (RCP) applications built using the Eclipse or NetBeans frameworks.

You need to package jinvoke.jar with your application, and add it to the classpath when running the application. There are no additional components to deploy, nor any need to bundle JNI DLLs, change the PATH, or modify java.library.path.

The jinvoke.jar file contains all the Java classes and native libraries required by J/Invoke. The first time the application is run, J/Invoke will extract the native library (jinvoke.dll on Windows) and store it alongside jinvoke.jar. This native library is loaded by J/Invoke at runtime, and is transparent to the user. The J/Invoke documentation provides walkthroughs for deployments employing Java Web Start and applets.