Programmer's Guide

Version: *1.2*
Last Updated: *February 21, 2003*

Contents

Introduction

In the few years since its first release, the Java™ programming language has grown immensely to become a popular platform. Many developers working on different platforms find their own advantages in using Java technology. One of them is of course the “write once, run anywhere” ability, allowing to write the software on one platform and run it on another.

Sometimes, however, Java programs have to interact with the native code. This is well justified by such reasons as performance, lack of features in the Java platform or legacy software interoperability. To solve this problem Java Native Interface (JNI) was introduced in the Java platform allowing programmers to write native code pieces and integrate them into their Java programs. The main difficulty arising from such approach is that the native code is completely disjoint from Java code in terms of browsing, debugging and maintenance.

In this document we introduce JNIWrapper – the product that allows to interface the native code retaining full control of the application on Java side at any level.

About This Guide

This guide introduces JNIWrapper, reviews its design goals, concepts and principles, provides the requirements for using JNIWrapper as well as sufficient information you need to know before starting working with the product.

This document covers all platform versions of JNIWrapper, and functions that teat a particular platform in a specific way, or specific configuration settings are marked accordingly.

New in This Version

New in this version (1.2) of the Programmer's Guide:

Added: “Function pointers” section

Added: “AWT Native Interface Support” section

Added: “Controlling Memory Allocation for Arrays” section

Updated: “Mapping Native Types to JNIWrapper Classes” table

Related Documents

The documents provided on the Documents page at the JNIWrapper site (http://www.jniwrapper.com) are aimed to help in understanding and effective usage of the JNIWrapper technology. Each JNIWrapper user should go through the documentation in the following order:

- Users should start with the current *Programmer's Guide* document describing the ideas and basics of the proposed software.

- Those who plan to develop their own software effectively should proceed from this Guide to *JNIWrapper Tutorial* providing detailed step-by-step instructions with useful code samples.
- Each version of JNIWrapper is supplied with an updated Release Notes document. Be sure to check the Release Notes before installing JNIWrapper to receive the up-to-date version specific information.
- You can also find useful information in the Frequently Asked Questions document that we regularly update basing on the questions we get from our users. The document can be accessed on-line at the JNIWrapper site.

About the JNIWrapper

Java™ is a very powerful platform, allowing programmers to develop the state-of-the-art software. It is, however, designed to run on a variety of different platforms and therefore does not encompass every feature of every platform. Certain basic things like creating a symbolic link under Linux or operating with registry items under Windows® are not supported in Java™. Programmers willing to do this are forced to write a native library and classes, interfacing with it, then debug the code using two different debuggers (Java- and native-side). These are sometimes difficult and always very time-consuming tasks. All of them can be avoided by using JNIWrapper – the Java library for calling native library functions. With JNIWrapper you can extensively use the potential of the underlying platform (like tray icons or custom shape splash screen) with only a single native library, having full control of the program flow on the Java side.

Technical Advantages

JNIWrapper has a number of technical advantages over the competition. Most important of them are:

- **Minimum behind-the-scene operation**. Users should always be able to see and understand what is happening when working with the native-side data. This helps both to develop and debug complex interactions between Java and native code.
- **High performance**. This has always been our priority. JNIWrapper has been specially tuned for performance, especially in cases where large amounts of data are involved in the interactions.
- **Automatic resource management**.
  - All resources allocated by JNIWrapper components are released automatically when no longer required. Users can treat JNIWrapper variables as usual object that can be picked by Java garbage collector.
  - JNIWrapper objects are also safe with regard to finalizers: all resources are guaranteed to be available during finalization.
- **Comprehensive native function invocation support.** JNIWrapper supports both `stdcall` and `cdecl` calling conventions and all complex C types including structures and unions. Callbacks are fully supported with

any parameter and return types and both calling conventions. For unexpected cases users can create their own types taking complete control over parameter behavior.

The product is extensively used in the projects carried out by the company, which ensures its efficiency, reliability, future support and improvement.

New in This Release

For detailed information about the changes in the current version be sure to check the `ReadMe.txt` inside the JNIWrapper package. The JNIWrapper changes history is also available on-line at http://www.jniwrapper.com/whatsnew.jsp.

Getting Started

System Requirements

The following are general requirements for running JNIWrapper on the following supported platform:

- **Windows**
  - **OS:** Windows 9x, Me, NT 4.0, 2000 and XP
  - **Java:** Java 2 SDK/JRE 1.3.x and onward
- **Linux**
  - **OS:**
  - **Java:** Java 2 SDK/JRE 1.4.x

There are no specific memory or other hardware requirements for developing an application based on JNIWrapper.

Other Platforms

Support for other platforms like Mac OS X, HP-UX, and Solaris is planned for the future.

Package Contents

JNIWrapper package consists of the following main files required for work:

- Library JAR file – `jniwrap.jar`
- Native code library
  - for Windows – `jniwrap.dll`
  - for Linux – `libjniwrap.so`
- License file –
  - for Windows – `jniwrap.lic`
  - for Linux – `jniwrap.lic`

All the files have to be placed in the appropriate locations. Please see the 'Configuring Software' section for more details about the product installation instructions. The package may also contain other files providing some useful information for the user, for example the `Readme.txt` file.

Configuring Software

As was stated above, JNIWrapper consists of three main files required for the software functioning: JAR file, native code library and the license file. The following paragraphs describe where each file should be located. No other configuration is required.

Library JAR File

The JNIWrapper JAR file should be located on the program class path. Due to the limitations of the Java native library loading mechanism it is not recommended to load JNIWrapper in custom class loaders unless you are sure that it will be loaded in only one such class loader.

Library file can also be placed on the boot class path or in the extension directory of Java runtime, but this is not required.

Native Code Library

The JNIWrapper native code library is loaded using the standard Java native code loading mechanism. Therefore the native code library file cannot be located inside a jar or some other package – it should always be a plain file on the file system. There are no known problems with placing the native code library file on a mapped drive or even using it from the network share using a UNC path. Do not rename the library file or it will not be loaded.

- Even though the native code library can be placed virtually anywhere, its actual location should be determined taking into account that Java code must find the library to load. It can be placed somewhere within the program library search path (value of the `java.library.path` system property, that is by default equal to the value of the system variable PATH on Windows or LD_LIBRARY_PATH on Linux).
- Alternatively, users can add a search path to the default library loader used by JNIWrapper or even write a custom one that searches for the native code in a predefined location. Using a default path may be preferable for development and library loader as a much better way for distributing a complete application.

Certain users may like to install the native code library into the directories on the default system path, for example:

- on Windows – the root of Windows installation or Windows\System32
- on Linux – <java_home>/lib/i386 or <java_home>/jre/lib/i386

Note that this requires having adequate access rights on the Windows NT/2000/XP and Linux systems. Installing the native code library using this way may be convenient, but is not a required procedure.

License File

Placing the license file is very simple – it just has to be located in the same directory as the native code library file. Do not rename the license file or it will not be recognized.

You may appear to have several licenses for JNIWrapper for different platforms supported. In this case you need to put all license key files as described above but preventing the file name conflict. JNIWrapper accepts multiple license files named like shown below:

- `jniwrap.lic`
- `jniwrap.lic1`
- `jniwrap.lic2`
- `...`
- `jniwrap.lic999`

Using JNIWrapper

Libraries and Functions

Finding Libraries

Since JNIWrapper accesses native libraries by name, there has to be some mechanism that finds an actual library file by its name. In JNIWrapper this is achieved by using library loaders. Library loaders are responsible for finding and loading a library given its name. Programmers can use the supplied implementation of a path-based library loader or create their own.

The latter approach requires implementation of the `com.jniwrapper.LibraryLoader` interface. This interface defines two methods: `findLibrary` for finding a file containing the required library by name, and `loadLibrary` for loading the library (by calling `System.load`), also by name. Suppose you have implemented a library loader called `MyLibraryLoader`. To make JNIWrapper look for the libraries using this loader you can write something like this:

```
Library.setDefaultLibraryLoader(new MyLibraryLoader());
```

Remember that the default loader specified this way is also used for loading the JNIWrapper native library.

Alternatively, you may use a custom loader to load a specific library only:

```
myLibrary.load(new MyLibraryLoader());
```

Using `DefaultLibraryLoader`

JNIWrapper comes with a convenient default implementation of the `LibraryLoader` interface – `DefaultLibraryLoader`. It looks for the libraries in a set of directories (path). The initial search path includes all directories from the `java.library.path` system property.

New directories can be added to the search path using the `addPath` method. `DefaultLibraryLoader` is a singleton and is also the one JNIWrapper uses by default.

If using the default path with some additions is sufficient, then adding directories to the search path is the only configuration needed. For example, if native libraries (or the JNIWrapper native code) are to be looked for in the Bin directory relative to the program working directory, the following line should be added to the JNIWrapper initialization part of your program:

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

Loading Libraries

Loading libraries does not require special attention if only one library loader is used, that is when the necessary library is found and loaded using the loader set by the `setDefaultLibraryLoader` method (or an instance of the

DefaultLibraryLoader class which is the initial value of that property).

If a custom library loader should be used, an explicit call to the library load method should be made before any function is loaded from that library.

Take a look at these examples. Using one library loader:

```
Library kernel32 = new Library("kernel32");
// Use kernel32 here
```

Using a custom library loader:

```
Library customLib = new Library("myCustomLib");
LibraryLoader myCustomLoader = new MyCustomLibraryLoader();
customLib.load(myCustomLoader);
// Use customLib here
```

Getting Functions

Functions are represented by the instances of com.jniwrapper.Function class. A function is always a part of some library and therefore cannot be instantiated on its own. To get a function from a library the getFunction should be used. For example:

```
Function getLastError = kernel32.getFunction("GetLastError");
```

Please remember that some compilers may mangle function names to include argument types or sizes. This version of JNIWrapper performs search only by the exact name, not by mangled names; therefore if you need to invoke a function with such a name you should specify the full mangled name. For example:

```
Function myFunc = myLibrary.getFunction("_myFunc@4");
```

Most libraries however export nice unmangled names. You can find out an exported function name by using tools such as *dumpbin* or *Dependency Checker*.

Calling Conventions

There are two widespread calling conventions: cdecl used in most C/C++ programs, and stdcall used, for example, in Pascal. Calling convention is a function property since a library can export functions that use different calling conventions. Use your library documentation to find out which calling convention is used by the functions you are interested in.

Calling Conventions on Windows

Most Windows® API functions use the stdcall calling convention. This is the default convention used by JNIWrapper. If the function you need to call uses a different calling convention you should set it in the function object. For example:

```
cdeclFunction.setCallingConvention(
       Function.CDECL_CALLING_CONVENTION);
```

Calling Conventions on Linux

The absolute majority of libraries on Linux use the `cdecl` calling convention. This is the default calling convention on Linux.

Parameters

In JNIWrapper parameters are passed to and from the native code using objects of the `com.jniwrapper.Parameter` type. These objects behave like variables of different types depending on the actual subclass used. All parameters are mutable, which means that users may change any value at any time. All parameters can be shared across function calls, i.e. one can pass the result of one function call directly to another without creating a new object. All parameters do their best in implementing a reasonable version of the `toString` method.

Primitive Types

JNIWrapper provides parameter classes for all the primitive types available in the native (C/C++) program: signed and unsigned integers of different sizes, floating point values, single-byte and wide characters, etc. Related types usually have a common superclass or implement a common interface. All simple types have a no-argument constructor that initializes a parameter to the default value (usually zero or equivalent), a constructor with the initial value and a copying constructor. They also have appropriately typed `getValue` and `setValue` methods. Take a look at the example of using the `DoubleFloat` parameter:

```
DoubleFloat d1 = new DoubleFloat(1.2345);
DoubleFloat d2 = new DoubleFloat(d1);
d1.setValue(9.876);
System.out.println(d2.getValue());
```

Structures and Unions

JNIWrapper supports C-like structures and unions. Like in C, the structure and union definitions are similar. Both provide a constructor that takes an array of parameters defining the structure or union contents. If you want to create a reusable Java class for a structure or unions, you can use a protected no-argument constructor and initialize the contents by calling the `init` method. This method is provided for convenience because class fields are initialized after the superclass constructor is invoked and therefore cannot be used as constructor arguments. Structure and union member values are accessed by directly querying the parameters specified in constructor for their values. For example:

```
/*
Structure definition in C:
struct SomeStruct
{
        int a;
        char b;
        double c;
}
*/
struct = new Structure(new Parameter[] {intField, charField,
      doubleField});
intField.setValue(10); // set struct.a value to 10
// invoke some code that modifies the structure here
System.out.println(doubleField); // prints new value of
struct.c
```

Structures can be defined with different alignments. Structure alignment is specified in the constructor (or in the call to the init method). Here is the example of a reusable structure definition which supports different alignments:

```
private static class TestStruct extends Structure {
    public Int _int = new Int();
    public Int8 _int8 = new Int8();
    public Int16 _int16 = new Int16();
    public Int32 _int32 = new Int32();
    public Int64 _int64 = new Int64();
    public LongInt _long = new LongInt();

    public TestStruct() {
        init(new Parameter[] {
                _int, _int8, _int16, _int32, _int64, _long
        });
    }

    public TestStruct(short alignment) {
        init(new Parameter[] {
                _int, _int8, _int16, _int32, _int64, _long
            }, alignment);
    }
}
```

Unlike C, in JNIWrapper user has to define the active union member explicitly using the `setActiveMember` method. However, this can be done after the function call where union data was modified. Take a look at the following examples of union usage:

```
union = new Union(new Parameter[] {intField, charField,
stringField, structField});
union.setActiveMember(stringField);
stringField.setValue(STRING_VALUE);
func1.invoke(result, union);
// ...
func2.invoke(union, (Parameter[])null);
union.setActiveMember(structField, true);
assertEquals(X_VALUE, structField.getX());
```

Structures and unions can consist of any simple or complex members that are JNIWrapper parameters. In the above example one of the union members is a structure.

Pointers

JNIWrapper supports C-like pointers to all the defined types. Users create a pointer by instantiating the `com.jniwrapper.Pointer` class. Pointers should always refer to some object. For example:

```
   Pointer pInt = new Pointer(new Int());
```

A pointer automatically allocates memory for its referenced object; it handles reads and writes, requiring the referenced object to read or write its data when the pointer itself is read or written. Thus, after any native function call all the parameters are updated even if they are referenced by several nested pointers. For example:

```
   Int value = new Int();
   Pointer ppInt = new Pointer(new Pointer(value));

   // invoke func(int **i) passing ppInt as a parameter
   func.invoke(null, ppInt);
   System.out.println(value) // value is updated!
```

In cases where the referenced object is read-only or write-only, one may use the `Pointer.Const` or `Pointer.OutOnly` types, respectively. Note, however, that using these classes cannot enforce the read-only or write-only policy on the native function which may still access the data inappropriately. It is recommended that these classes are only used for performance improvements.

JNIWrapper supports pointers that reference undefined values (`void*` in C) through the `Pointer.Void` class. Use this class when you do not care about the actual referenced object and do not have to allocate the memory a pointer points to. For example, if you need a constant `(HWND)-1` you can use the following construct:

```
   Pointer.Void HWND_TOPMOST = new Pointer.Void(-1);
```

`Pointer.Void` is not a pointer: it does not have a referenced object and it is not assignable to and from any other kind of pointer. The name of the type only reflects the fact that this parameter always has the same size as platform-dependent pointer does.

Function Pointers

Another capability of the `Pointer.Void` class is that it can also be used to represent a function pointer and to call the function it points to. The asFunction() method returns a function object that can be used to invoke a function that a given pointer points to.

Consider the following example. A native library provides a function to install a callback that returns an old callback function pointer:

```
   typedef void (*PCallbackType)(int);
   PCallbackType installCallback(PCallbackType);
```

One may install a hook to monitor callback invocation as follows:

```
   // Field declaration
```

```
Pointer.Void myOldCallback;
// ...
// Callback installation code
installCallback.invoke(myOldCallback, new HookCallback());
```

and later inside the HookCallback class

```
protected void callback() {
    // do some hook stuff
    myOldCallback.asFunction().invoke(null, intParam);
}
```

Limited pointer arithmetics is supported through the ArithmeticalPointer class. This pointer also manages one referenced object, but also accommodates an offset from its initial value. Such pointer can be used for passing to the functions such as strtok that offset a pointer to iterate through data. Note that the referenced object still cannot be changed and is always read and written at its initial offset.

Arrays

JNIWrapper supports two types of arrays: primitive arrays, that are made of primitive values such as integers or characters and complex arrays, that can consist of the elements of any implemented type. The former can be basically implemented using the latter, but for the primitive types primitive arrays are more efficient.

Arrays are represented by the instance of com.jniwrapper.PrimitiveArray and com.jniwrapper.ComplexArray for the primitive and complex arrays, respectively. The main difference between the two types of arrays is that a primitive array is a plain data block that contains sequential data of a given type, while a complex array is a sequential storage of elements of any complexity that are all read and written whenever an array is read or written, respectively. This means that an array of pointers cannot be implemented as a primitive array because the referenced objects will not be written or read when needed.

The simplest method of creating an array is using a constructor specifying a sample parameter and array size. Note, that array member type should be correctly cloneable. For example, to create an array of bytes you can use the following construct:

```
PrimitiveArray val = new PrimitiveArray(new Int8(), 256);
```

Another method of creating an array is using a Java array of parameters that should constitute it. This is achieved by using a constructor taking a Parameter[] argument. Both primitive and complex arrays can be created this way. Here is an example of creating an array of pointers to integers:

```
Parameter[] members = new Parameter[10];
for(int i = 0; i < 10; i++) {
       members[i] = new Pointer(new Int(i));
}
ComplexArray result = new ComplexArray(members);
```

When using arrays one should always remember that sometimes arrays are stored or passed to functions not as plain data, but as a pointer to the array contents. The most typical case is when an array argument or member is defined as a pointer to type (e.g. double*). In this case the actual passed parameter should be a Pointer. For example:

```
/*
C declaration:
struct s
{
       int size;
       double *data;
}
*/
Int intMember = new Int(50);
PrimitiveArray arrayMember = new
PrimitiveArray(DoubleFloat.class, 50);
Structure s = new Structure(new Parameter[] {intMember,
        new Pointer(arrayMember)});
```

If in the previous example the second member is declared as double data[50] (predefined size array), the pointer wrapper should not be used.

Controlling Memory Allocation for Arrays

In most cases the array sizes are known or may be computed before a call is made. There are cases, however, when deciding array size before a function call is not possible or not efficient. In these cases the array that is passed by pointer is either resized to accommodate all the data or allocated altogether by the callee. In the first case the caller is usually still responsible for memory deallocation while in the second the memory management is most likely the responsibility of the callee. The common thing in both cases is that the called function returns the new size of the array as one of the results of the call. JNIWrapper supports both ways of required memory management by using the special array pointers – ResizingPointer and ExternalArrayPointer. Each of these pointers does not read the array it points at after the function call. The array should be read after the call is complete using the readArray(int count) method of the pointer. For example:

```
PrimitiveArray myArray = new PrimitiveArray(Int8.class,
length);

Int16 len = new Int16(length);
ExternalArrayPointer pArray = new ResizingPointer( myArray);
Function func = getFunction("myFunction");
func.invoke(null, pArray, new Pointer(len));

length = (short)len.getValue();
pArray.readArray(length);
// use  myArray here
```

The rule of thumb for choosing the correct pointer is: if you want JNIWrapper to manage the memory allocated for the array – choose ResizingPointer, if you do not need the memory management for the array memory – choose `ExternalArrayPointer`.

Strings

Strings in JNIWrapper are character arrays of a predefined size with convenience methods for getting and setting string values as zero terminated strings. JNIWrapper supports two types of strings: single-byte, or ANSI strings, and Unicode strings.

All strings are defined with a maximum length that a string value can occupy. It is illegal to set the string parameter value to a value longer than its maximum length. Passing string argument to a function that writes more data than is allocated may result in an error just as it does for the native programs. On the other hand, strings are safe in a way that the data is parsed until the terminating zero (of the appropriate length) is found or maximum string length is reached. Therefore even a bad string without a terminating zero will not cause memory access failures in your program. Here are the examples of string usage:

```
   AnsiString s = new AnsiString("Hello, World!");
   WideString s2 = new WideString(20);
   s2.setValue("Goodbye, World!");
```

Similarly to arrays, strings in the native code can be both pointers to string data and character arrays themselves. When using strings as structure members, use the same guidelines to determine whether a pointer wrapper should be used. When passing strings as function arguments, however, strings are always passed as pointers and JNIWrapper does pointer wrapping automatically. You should remember though that wrapping is just a convenience feature and passing a `char**` will require creating a new `Pointer(new Pointer(new AnsiString()))`.

Mapping Native Types to JNIWrapper Classes

Below is given the mappings table for most commonly used data types along with some comments.

Native Type (C/C++)	JNIWrapper type	Comments
Boolean Types		
bool	Bool	
Character types		
char	Char	
wchar_t	WideChar	

Native Type (C/C++)	JNIWrapper type	Comments
Integer types		
short	ShortInt	The unsigned types are represented by prepending U to the type name, e.g. unsigned int (or unsigned) type is UInt. There are also types for predefined-width integers: Int8, Int16, Int32 and Int64, they also have the unsigned variants.
int	Int	
long	LongInt	
Floating-point types		
float	SingleFloat	
double	DoubleFloat	
long double	LongDouble	Long double is the same as double (8-byte floating-point value) on win32 platform.
Pointer types (not arrays)		
void *	Pointer.Void	To create a pointer to a value (variable) of a known type use Pointer class. For example: `int *i;` is `Pointer i = new Pointer(new Int());` Use `Pointer.Const` if the referenced value is not modified by the calling function, this includes modifications of anything that is referenced by that value. Use `Pointer.OutOnly` if the referenced value is not read by the calling function. Function pointers can be void * if the referenced value is not of interest to the program or a subclass of the Callback if the pointer should be to the user-provided callback function.
char *	AnsiString	
wchar_t *	WideString	
Arrays		

Arrays are represented by types PrimitiveArray and ComplexArray parametrized by parameters that represent the actual type. For example:
`int i[10];`
is
`PrimitiveArray i = new PrimitiveArray(Int.class, 10);`
The difference between primitive and complex array types is that PrimitiveArray can be only of primitive types (int, char, float) and not of arrays, pointers, structures, etc. ComplexArray has no restriction on its element type.
Sometimes arrays can be specified as pointers in function signature, for example: `void foo(int *arg);` but if the actual value is a pointer to some number of integers - use array as a parameter.		

Native Type (C/C++)	JNIWrapper type	Comments
Structures and unions		
struct	Structure	
union	Union	
Function pointers		
To create an object callable from the native code use the Callback class. To call a function returned from the native code use the method asFunction of the Pointer.Void class.		

Windows API includes many data types which are not listed here (e.g. DWORD, HANDLE), if you need to use one of such types use Windows-specific documentation such as MSDN to find out the actual C type that corresponds to it (e.g. LPSTR corresponds to char*) and use the relevant JNIWrapper type for the argument. You can also check the on-line 'Windows Data Types' table available at http://www.jniwrapper.com/wintypes.jsp.

Calling Functions

Functions are called using the invoke method of the Function class. The arguments and return value are specified using variables of the Parameter type. There are several overloaded versions of the invoke method, but the idea and argument structure is similar: the first argument is always a holder for the return value and the rest are arguments in the order they appear in the function declaration. When a function is called, all passed parameters are passed to it and then the returned value is stored in its placeholder. It is allowed to pass null instead of the return value parameter, in which case it will be ignored, but it is allowed for primitive types only. Doing this when the return value is not of a primitive type may result in an error, because memory must be allocated by the function caller if the function return value is big and there is no way to allocate appropriate structure automatically without knowing the actual return value type. There are no restrictions on the argument or return value types as long as they are what a function expects.

Calling convention must be set before the function is first called, failure to do so will result in an error. Here is a complete example of calling a function:

```
Function sprintf =
       new Library("msvcrt").getFunction("sprintf");
sprintf.setCallingConvention(
       Function.CDECL_CALLING_CONVENTION);
AnsiString result_buffer = new AnsiString();
sprintf.invoke(
       null,
       result_buffer,
       new AnsiString("Hello, %s!"),
       new AnsiString("World"));
```

```
    System.out.println("result = " + result_buffer.getValue());
    //Output: result = Hello, World!
```

This example shows that there is no problem with calling functions even with variable argument number.

JNIWrapper checks that stack is left in a consistent state after a function is invoked, and handles most of the failures by throwing a Java exception. All exceptions descend from `com.jniwrapper.FunctionExecutionException`.

Function class provides a shortcut to call a function from a library without creating `Library` and `Function` instances – the `call` method. To use it, write the following:

```
Function.call("user32.dll",
       "MessageBeep", retVal, new UInt(1));
```

```
 Function.call("/lib/libc.so.6",
    printf", null, new AnsiString("Hello, World!\n"));
```

This will load a library using the default loader, lookup a function and invoke it using the default calling convention. It is not recommended that this method is used when your program makes a lot of native function calls, because it is more resource intensive. However, it is perfectly fine to use it in simple cases like getting the current directory.

Callbacks

Callback is a user-defined function that is called by the library code at the appropriate time. Callbacks can have arguments and return a value. JNIWrapper supports callbacks with any kind of arguments and return values, as well as `stdcall` and `cdecl` calling conventions. Callbacks are represented by subclasses of the `com.jniwrapper.Callback` class.

Passing callback as an argument is no different from passing any other value – you just have to put an instance of the `Callback` class as the corresponding argument. For example:

```
final class EnumWindowsProc extends Callback
// ...
EnumWindowsProc enumWindowsProc = new EnumWindowsProc();
Function.call("user32.dll", "EnumWindows", retVal,
enumWindowsProc, new Int32(57));
```

```
class MyComparator extends Callback
// ...
Function.call("/lib/libc.so.6", "qsort", null,
       dataPointer, new Int(size), new Int(memberLen),
       new MyComparator());
```

When a callback is no longer needed, user should explicitly free the resources associated with it by invoking its dispose method. This is the only case in

JNIWrapper where resources are to be explicitly freed, because some types of the callback objects may have no references from the code and still be active, for example a window procedure callback instance may be created once when the window class is registered and remain active during the entire program life-cycle.

To implement a callback user has to subclass a `Callback` class implementing the `callback` abstract method . The callback arguments and return value are specified in the constructor (by calling either superclass constructor with parameters or `init` method). The order of parameters is the same as for the invoke method of the class `Function`: first goes the return value, and then the arguments. Calling convention should also be set in the constructor if it is different from the default one.

When a `callback` method is invoked, argument parameters contain the values of the passed arguments; after a `callback` method completes, the value of the return value parameter is returned to the caller. Here is an example of a callback that takes an integer argument and returns its value incremented by one:

```
/*
C callback declaration:
int callback(int);
*/
private static class IncIntCallback extends Callback {
    private Int _arg = new Int();
    private Int _result = new Int();

    public IncIntCallback() {
        init(new Parameter[] {_arg}, _result);
    }

    public void callback() {
        _result.setValue(_arg.getValue() + 1);
    }
}
```

Multi-threading

JNIWrapper was designed with threading in mind and is already successfully used in a multi-threaded environment. The following sections describe thread safety of JNIWrapper components.

Parameters

Parameters in JNIWrapper are not synchronized, because synchronizing at this level is very inefficient, and performance was a high priority goal in JNIWrapper development. Therefore, the access to parameter data should be enclosed in synchronization blocks if there is a possibility for more than one thread to access the data at a time. If you have only one parameter that is shared between threads you can synchronize on that objects monitor:

```
Int32 sharedInt = new Int32();
```

```
   // ...
   // First thread
   synchronized(sharedInt) {
       someFunction.invoke(sharedInt);
   }
   //...
   // Second thread
   synchronized(sharedInt) {
       if (sharedInt.getValue() == 10)
           System.out.println("Value is set to 10");
```

Functions

Function invocation in JNIWrapper is completely thread-safe. If the called function is thread-safe, any number of threads may invoke it concurrently at any time. No synchronization is required and/or performed, and calls are executed fully concurrently as if invoking simple Java methods.

Callbacks

In this version callbacks are not thread-safe and not reentrant.

AWT Native Interface Support

Java™ includes the cross-platform standard windowing library called Abstract Window Toolkit(AWT). One of the design principles of the AWT was to include only the features that can be implemented on all platforms targeted for Java™. This imposed some limitations on the windowing interface features. To help users access the native controls that stand behind the AWT the AWT native interface (JAWT) was introduced. JNIWrapper also supports this interface so that users do not need to write native code to access the JAWT features.

Using JAWT Support

On the native side all the JAWT functionality can be accessed through several structures defined in the `include/jawt.h` file in the JDK directory. The root structure – `JAWT` – is available from the function `GetAWT`. JNIWrapper provides the class `com.jniwrapper.util.JAWT` that implements all the functionality available through the `JAWT` structure. The classes `JAWT_DrawingSurface` and `JAWT_DrawingSurfaceInfo` correspond to the structures with the same names in the `jawt.h` and provide the same functionality.

Accessing Native Controls Data

The most common use of the JAWT interface is to get a handle of the native control that corresponds to the given component. This data is returned in the platform-dependent structure pointed to by the `platformInfo` member of the `jawt_DrawingSurfaceInfo` structure. The contents of this structure are defined in the `jawt_md.h` file in the `include/<platform>/` directory of the JDK installation. The correct structure has to be passed to the constructor of the `JAWT_DrawingSurfaceInfo` class. A sample structure for Win32 is shown below:

```
public class Win32DSI extends Structure
{
    private Pointer.Void _handle = new Pointer.Void();
    private Pointer.Void _hdc = new Pointer.Void();
    private Pointer.Void _hpalette = new Pointer.Void();

    public Win32DSI()
    {
        init(new Parameter[]{_handle, _hdc, _hpalette}, (short)
8);
    }

    /**
     * Returns target component handle (either window or bitmap
handle).
     */
    public Pointer.Void getHandle()
```

```
        {
            return _handle;
        }

        /**
         * Retruns DC handle. This handle should be used for
    drawing instead of handles returned
         * from the <code>GetDC</code> or <code>BeginPaint</code>.
         */
        public Pointer.Void getHdc()
        {
            return _hdc;
        }

        /**
         * Returns palette handle.
         */
        public Pointer.Void getHpalette()
        {
            return _hpalette;
        }
    }
```

Platform-dependent data structures are currently not included in the main JNIWrapper distribution.

Getting HWND of a Window – an Example

Below is the example code snippet that gets a Win32 window handle of an AWT window. It is very similar to the C-code required to do the same thing.

```
JAWT_DrawingSurface ds = JAWT.getDrawingSurface(window);
ds.lock();
Win32DSI win32DSI = new Win32DSI();
JAWT_DrawingSurfaceInfo dsi = new
         JAWT_DrawingSurfaceInfo(win32DSI);
Pointer pDsi = new Pointer(dsi);
ds.getDrawingSurfaceInfo(pDsi);
int result = (int) win32DSI.getHandle().getValue();
ds.freeDrawingSurfaceInfo(pDsi);
ds.unlock();
JAWT.freeDrawingSurface(ds);
return result;
```

JAWT Support in Different JDK Versions

When programming using JAWT interface it should be kept in mind that JAWT is a relatively new technology. It was introduced only in JDK 1.3 with limited functionality, has had many improvements to the JDK version 1.4, and JAWT is intended to completely replace such native-related functions as `sun.awt.windows.WToolkit.getNativeWindowHandleFromComponent`. For the JDK 1.3 one should expect very limited support for JAWT. For example the argument of the `getDrawingSurface` method can be only a `java.awt.Canvas`. JDK 1.4 introduced new functions such as AWT locking and unlocking as well as extended the capabilities of the existing ones.

Support

If you encounter any problems or have questions regarding our product, please check the documents listed below. The answer to your question may already be listed there:

- Installation instructions;
- User Guide;
- Frequently Asked Questions (FAQ) page at the following address:

  http://www.jniwrapper.com/support.jsp.

If none of the above sources contains the required information, please e-mail us at: support@jniwrapper.com.

Reporting Problems

Should you experience any problem or find any bugs, please submit the issue to us using the special report form on the JNIWrapper site at the following address:

http://www.jniwrapper.com/support_form.jsp

The form will help you provide all the necessary information.

Troubleshooting

For finding a solution on your problem please visit the Troubleshooting page on our site at the following address:

http://www.jniwrapper.com/tshoot.jsp

This page is regularly updated using information from the support requests.

If you don't find a solution please e-mail us at support@jniwrapper.com or report the problem as described in previous section.

Where to Get New Version

To obtain the latest version of JNIWrapper and to receive up-to-date information please visit: http://www.jniwrapper.com

Alphabetical Index