**JNA API 5.2.0**

**JNA API Documentation**

Java Native Access This document is the API specification for the [JNA](https://github.com/java-native-access/jna) library for simplified native library access for Java.

**See:**[**Description**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#overview.description)

|  |  |
| --- | --- |
| **Java Native Access** | |
| **Package** | **Description** |
| [**com.sun.jna**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/package-summary.html) | Provides simplified native library access. |
| [**com.sun.jna.ptr**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/ptr/package-summary.html) | Provides various native pointer-to-type (<type> \*) representations. |
| [**com.sun.jna.win32**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/win32/package-summary.html) | Provides type and function mappers required for standard APIs on the Windows platform. |
| **Platform Utilities** | |
| **Package** | **Description** |
| [**com.sun.jna.platform**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/package-summary.html) | Provides cross-platform utilities based on platform-specific libraries. |
| [**com.sun.jna.platform.dnd**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/dnd/package-summary.html) | Provides integrated, extended drag and drop functionality, allowing ghosted drag images to be used on all platforms. |

|  |  |
| --- | --- |
| **Platform Specific** | |
| **Package** | **Description** |
| [**com.sun.jna.platform.linux**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/linux/package-summary.html) | Provides common library mappings for Linux. |
| [**com.sun.jna.platform.mac**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/mac/package-summary.html) | Provides common library mappings for the OS X platform. |
| [**com.sun.jna.platform.unix**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/unix/package-summary.html) | Provides common library mappings for Unix and X11-based platforms. |
| [**com.sun.jna.platform.unix.solaris**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/unix/solaris/package-summary.html) | Provides common library mappings for the Solaris (SunOS) platform. |
| [**com.sun.jna.platform.win32**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/package-summary.html) | Provides common library mappings for the Windows platform. |
| [**com.sun.jna.platform.win32.COM**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/COM/package-summary.html) | Provides... |
| [**com.sun.jna.platform.win32.COM.tlb**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/COM/tlb/package-summary.html) | Provides... |
| [**com.sun.jna.platform.win32.COM.tlb.imp**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/COM/tlb/imp/package-summary.html) | Provides... |
| [**com.sun.jna.platform.win32.COM.util**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/COM/util/package-summary.html) |  |
| [**com.sun.jna.platform.win32.COM.util.annotation**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/win32/COM/util/annotation/package-summary.html) |  |
| [**com.sun.jna.platform.wince**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/platform/wince/package-summary.html) |  |

**Java Native Access**

This document is the API specification for the [JNA](https://github.com/java-native-access/jna) library for simplified native library access for Java.

[**Top**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#navbar_top)[**JNA Home**](https://github.com/java-native-access/jna)[**API w/FRAMES**](http://java-native-access.github.io/jna/5.2.0/javadoc/index.html)

**Java Native Access (JNA)**

JNA provides simplified access to native library methods without requiring any additional JNI or native code.

**Table of Contents**

* [**Loading JNA**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#loading)
* [**Library Mapping**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#library-mapping)
* [**Function Mapping**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#function-mapping)
* [**Type Mapping**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#marshalling)
  + [**Primitive Types**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#marshalling)
  + [**Pointers**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#pointers)
  + [**Strings**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#strings)
  + [**Wide (UNICODE) Strings**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#wide-strings)
  + [**Primitive Arrays**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#arrays)
  + [**Buffers/Memory Blocks**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#buffers)
  + [**Callbacks/Function Pointers**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#callbacks)
  + [**Variable Argument Lists (Varargs)**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#varargs)
  + [**Structures**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#structures)
  + [**Unions**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#unions)
  + [**Java Objects**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#java-objects)
  + [**Last Error**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#last-error)
* [**Invocation Mapping**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#invocation-mapping)
* [**Library Global Data**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#global-data)
* [**VM Crash Protection**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#crash-protection)
* [**Performance**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#performance)

**Loading JNA**

JNA includes a small, platform-specific shared library which enables all native access. When the [Native](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html) class is first accessed, JNA will first attempt to load this library from the directories specified in jna.boot.library.path. If that fails and jna.nosys=false is set, it will fall back to loading from the system library paths. Finally it will attempt to extract the stub library from from the JNA jar file, and load it.

The jna.boot.library.path property is mainly to support jna.jar being included in -Xbootclasspath, where java.library.path and LD\_LIBRARY\_PATH are ignored. It is also useful for designating a version of the library to use in preference to any which may already be installed on the system.

Loading from the system may be enabled by jna.nosys=false, and unpacking from the jar file may be disabled by jna.nounpack=true.

The library name used to search for JNA's native library may be altered by setting jna.boot.library.name, which defaults to "jnidispatch". It may be useful to set this value if your system requires unique names for shared libraries (rather than unique paths), or if your system must store different versions of the JNA shared library (e.g. for different architectures) in the same directory.

**Library Mapping**

When you've determined which shared library holds the methods to which you need access, create a class corresponding to that library. For example, a mapping for the C library itself would look like one of the following:

// Alternative 1: interface-mapped class, dynamically load the C library

public interface CLibrary extends Library {

CLibrary INSTANCE = (CLibrary)Native.load("c", CLibrary.class);

}

// Alternative 2: direct-mapped class (uses a concrete class rather than an interface, with a slight variation in [method declarations](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#direct-mapping)).

public class CLibrary {

static {

Native.register("c");

}

}

The String passed to the [Native.load(String,Class)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html#load-java.lang.String-java.lang.Class-) (or [NativeLibrary.getInstance(String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html#getInstance-java.lang.String-)) method is the undecorated name of the shared library file. Here are some examples of library name mappings.

|  |  |  |
| --- | --- | --- |
| **OS** | **Library Name** | **String** |
| Windows | user32.dll | user32 |
| Linux | libX11.so | X11 |
| Mac OS X | libm.dylib | m |
| Mac OS X Framework | /System/Library/Frameworks/Carbon.framework/Carbon | Carbon |
| Any Platform | <current process> | null |

Any given native library with a unique filesystem path is represented by a single instance of [NativeLibrary](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html" \o "class in com.sun.jna) and obtained via [NativeLibrary.getInstance(String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html" \l "getInstance-java.lang.String-). The native library will be unloaded when no longer referenced by any Java code.

If the library name is null, your mappings will apply to the current process instead of a separately loaded library. This may help avoid conflicts if there are several incompatible versions of a library available.

The search path for loaded native libraries may be modified by setting jna.library.path and a few other properties. You may also bundle native libraries in a jar file and have JNA automatically extract them for loading. See [NativeLibrary](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html" \o "class in com.sun.jna) for details.

**Function Mapping**

Function names are mapped directly from their Java interface name to the symbol exported by the native library. For instance, the function to convert an ASCII string into an integer would look like this:

public interface CLibrary extends Library {

int atol(String s);

}

Alternatively, you can map directly to a declared native method (with [some restrictions](https://github.com/java-native-access/jna/blob/master/www/DirectMapping.md)):

public class CLibrary {

public static native int atol(String s);

}

If you prefer to rename the Java methods to conform to Java coding conventions, then you can provide an entry ([Library.OPTION\_FUNCTION\_MAPPER](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Library.html" \l "OPTION_FUNCTION_MAPPER)/[FunctionMapper](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/FunctionMapper.html" \o "interface in com.sun.jna)) in the options Map passed to [Native.load()](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "load-java.lang.String-java.lang.Class-java.util.Map-) which maps the Java names to the native names. While this keeps your Java code a little cleaner, the additional mapping of names may make it a little less obvious the native functions being called.

An instance of the [Function](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Function.html) class is obtained through the [NativeLibrary](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html" \o "class in com.sun.jna) instance corresponding to the containing native library. This [Function](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Function.html) instance handles argument marshalling and delegation to the native function.

**Marshalling/Unmarshalling (Java/Native Type Conversions)**

Java types must be chosen to match native types of the same size. Following are the types supported by the JNA library.

|  |  |  |
| --- | --- | --- |
| **C Type** | **Native Representation** | **Java Type** |
| char | 8-bit integer | byte |
| wchar\_t | platform-dependent | char |
| short | 16-bit integer | short |
| int | 32-bit integer | int |
| int | boolean flag | boolean |
| enum | enumeration type | int (usually) |
| long long, \_\_int64 | 64-bit integer | long |
| float | 32-bit floating point | float |
| double | 64-bit floating point | double |
| pointer (e.g. void\*) | platform-dependent (32- or 64-bit pointer to memory) | Buffer [Pointer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html) |
| pointer (e.g. void\*), array | 32- or 64-bit pointer to memory (argument/return) contiguous memory (struct member) | <P>[] (array of primitive type) |
| In addition to the above types, which are supported at the native layer, the JNA Java library automatically handles the following types. All but NativeMapped and NativeLong are converted to [Pointer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html) before being passed to the native layer. | | |
| long | platform-dependent (32- or 64-bit integer) | [NativeLong](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLong.html) |
| const char\* | NUL-terminated array (native encoding or jna.encoding) | String |
| const wchar\_t\* | NUL-terminated array (unicode) | [WString](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/WString.html) |
| char\*\* | NULL-terminated array of C strings | String[] |
| wchar\_t\*\* | NULL-terminated array of wide C strings | [WString[]](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/WString.html) |
| void\*\* | NULL-terminated array of pointers | [Pointer[]](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html) |
| struct\* struct | pointer to struct (argument or return) ([or explicitly](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByReference.html)) struct by value (member of struct) ([or explicitly](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByValue.html)) | [Structure](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html) |
| union | same as Structure | [Union](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Union.html) |
| struct[] | array of structs, contiguous in memory | [Structure[]](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html) |
| void (\*FP)() | function pointer (Java or native) | [Callback](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Callback.html) |
| pointer (<T> \*) | same as Pointer | [PointerType](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/PointerType.html) |
| other | integer type | [IntegerType](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/IntegerType.html) |
| other | custom mapping, depends on definition | [NativeMapped](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeMapped.html) |

**NOTES**

* Unsigned values may be passed by assigning the corresponding two's-complement representation to the signed type of the same size.
* Java arrays of primitive type may be wrapped by Buffer in order to access a subset of the array (changing the effective size and/or offest).
* Java arrays of primitive type and non-direct Buffers are only valid for use within the scope of a single call. If the native code keeps a reference to the memory, use [Memory](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Memory.html) or direct ByteBuffers instead.
* Primitive arrays and structures as members of a structure are overlaid on the parent structure memory.
* Bitfields must be manually packed into an integer type.
* All other types must eventually be converted to one of the types in this table. Methods with arguments or return values of types other than these must either use types deriving from [NativeMapped](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeMapped.html" \o "interface in com.sun.jna) or supply type conversion information for the unsupported types.
* Type mapping behavior may be customized by providing a [TypeMapper](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/TypeMapper.html" \o "interface in com.sun.jna) for the [Library.OPTION\_TYPE\_MAPPER](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Library.html" \l "OPTION_TYPE_MAPPER) option when initializing a library interface. See [W32APITypeMapper](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/win32/W32APITypeMapper.html) for an example which provides custom conversion of boolean and String types. You are free to use whatever types are convenient in your defined interfaces, but all custom types *must* provide a mapping to one of the basic or derived types listed above.
* Type mapping may also be customized on a per-class basis for user-defined types by making the user-defined type implement the [NativeMapped](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeMapped.html" \o "interface in com.sun.jna) interface.
* Structure and Union are *not* converted to Pointer when passed by value.

***Primitive Arrays***

Java primitive arrays may be used wherever a native primitive array is used. Any changes made by the native code to an array during a function call will be reflected in the Java array. **If the native code will use the array outside of the function call where the array is provided,**[**Memory**](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Memory.html)**or Buffer should be used instead。**

To map a native multi-dimensional array, use a single-dimensional Java array with a number of elements equivalent to the full native array, e.g.

// Original C code

#define DIM0 2

#define DIM1 3

int array[DIM0][DIM1];

int i,j;

for (i=0;i < DIM0;i++) {

for (j=0;j < DIM1;j++) {

array[i][j] = i\*DIM1 + j;

}

}

// Equivalent JNA code

final int DIM0 = 2;

final int DIM1 = 3;

int[] array = new int[6];

for (int i=0;i < DIM0;i++) {

for (int j=0;j < DIM1;j++) {

array[i\*DIM1 + j] = i\*DIM1 + j;

}

}

***Pointers***

Pointers may be used as an opaque type from which other data types may be extracted. The Pointer type is a reasonable fallback for any pointer-based type (including arrays). The user is generally not allowed to construct a Pointer de novo.

Type-safe pointers may be defined by deriving from the [PointerType](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/PointerType.html" \o "class in com.sun.jna) class. Any such user-defined type will be treated the same as a [Pointer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html).

***Strings***

Java Strings perform the same function as the native types const char\* and const wchar\_t\* (NUL-terminated arrays). In order to use the proper type when calling a native function, we have to introduce some sort of annotation to identify how the java String should be converted. Java Strings are normally converted to char\* since this is the most common usage of strings. Strings are automatically converted to a NUL-terminated array of char across the function call. Returned char\* values are automatically copied into a String if the method signature returns String (strdup, for example).

If the native method returns char\* and actually allocates memory, a return type of [Pointer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html) should be used to avoid leaking the memory. It is then up to you to take the necessary steps to free the allocated memory.

When converting Java unicode characters into an array of char, the default platform encoding is used, unless the system propertyjna.encoding is set to a valid encoding. This property may be set to "UTF8", for example, to ensure all native strings use that encoding.

Arrays of String passed to native code (either as a function argument or callback return value) will be converted into a NULL-terminated array of char\* (or wchar\_t\* in the case of an array of WString.

***Wide Strings***

The [WString](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/WString.html" \o "class in com.sun.jna) class is used to identify wide character strings. Unicode values are copied directly from the Java char array to a native wchar\_t array.

***Buffers/Memory Blocks***

Use arrays to represent buffers of primitive types passed to a function for use only during the function invocation. If the native code keeps a pointer to the memory after the native function returns, use direct ByteBuffers or [Memory](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Memory.html) instead.

A native method cannot return a Java array, since there is no canonical way to indicate the intended length of the returned array. Instead, use one of the array access methods in the Pointer class, supplying the length of the returned array.

Buffers may also be used as a memory buffer input argument; direct byte buffers can often provide much improved performance over primitive arrays. A pointer provided by native code may be converted to a Buffer by calling [Pointer.getByteBuffer(long, long)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html" \l "getByteBuffer-long-long-).

If you need to pass in a subset of a primitive array, you can do so by wrapping it in a Buffer subclass, such as ByteBuffer, using the ByteBuffer.wrap(byte[],int,int) method. Wrapping an array in a buffer also allows you to pass only a subset of a Java array to the native function.

***Callbacks (Function Pointers)***

JNA supports supplying Java callbacks to native code. You must define an interface that extends the [Callback](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Callback.html) interface, and define a single callback method with a signature that matches the function pointer required by the native code. The name of the method may be something other than "callback" only if there is only a single method in the interface which extends Callback or the class which implements [Callback](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Callback.html). The arguments and return value follow the same rules as for a direct function invocation.

When accessing Windows APIs, sometimes the documentation indicates that a function pointer parameter must refer to a function that resides in a DLL. In these instances, add the [DLLCallback](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/win32/DLLCallback.html" \o "interface in com.sun.jna.win32) interface to your callback definition. The function pointer as seen by Windows will be located in the jnidispatch.dll module.

If the callback returns a String or String[], the returned memory will be valid until the returned object is GC'd.

If your native code initializes function pointers within a struct, JNA will automatically generate a Callback instance matching the declared type. This enables you to easily call the function supplied by native code using proper Java syntax.

// Original C code

struct \_functions {

int (\*open)(const char\*,int);

int (\*close)(int);

};

// Equivalent JNA mapping

public class Functions extends Structure {

public static interface OpenFunc extends Callback {

int invoke(String name, int options);

}

public static interface CloseFunc extends Callback {

int invoke(int fd);

}

public OpenFunc open;

public CloseFunc close;

}

...

Functions funcs = new Functions();

lib.init(funcs);

int fd = funcs.open.invoke("myfile", 0);

funcs.close.invoke(fd);

Callbacks may also be used as return values. Native function pointers are wrapped in a proxy implementing the declared Callback type, to facilitate calling from Java.

// Original C code

typedef void (\*sig\_t)(int);

sig\_t signal(int signal, sig\_t sigfunc);

// Equivalent JNA mapping

public interface CLibrary extends Library {

public interface SignalFunction extends Callback {

void invoke(int signal);

}

SignalFunction signal(int signal, SignalFunction func);

}

If you need control over the thread context in which a Callback operates, you can install a [CallbackThreadInitializer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/CallbackThreadInitializer.html" \o "class in com.sun.jna) for any given callback object. The first time the callback is called on a thread that is not currently attached to the VM, the initializer will be queried to determine how the thread should be set up. You can indicate the desired name, thread group, and daemon state for the thread, as well as indicating whether the thread should be left attached to the VM after callback exit. The latter improves performance if you know you will be getting multiple callbacks on the same thread, avoiding the need for the VM to generate multiple Java Thread objects for the same native thread. If you do leave the native thread attached, you should either ensure you detach it at some later point (by calling [Native.detach(boolean)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "detach-boolean-) from within the callback just prior to return) or return true from your[CallbackThreadInitializer.isDaemon(Callback)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/CallbackThreadInitializer.html#isDaemon-com.sun.jna.Callback-) method so that the native thread will not prevent the VM from exiting.

If you don't need to otherwise customize the callback thread, you can simply call [Native.detach(boolean)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "detach-boolean-) from within your callback to indicate whether the thread attachment should be maintained or not.

***Varargs***

The C varargs function definition may be mapped to a Java varargs method definition. For example,

// Original C code

extern int printf(const char\* fmt, ...);

// Equivalent JNA mapping

interface CLibrary extends Library {

int printf(String fmt, ...);

}

*Varargs are not supported when using*[*Direct mapping*](https://github.com/java-native-access/jna/blob/master/www/DirectMapping.md)*.*

***Structures***

The Java [Structure](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html) represents a native struct. By default, this type is treated as a pointer to structure (struct \*) on the native side when used as a parameter or return value. **When used as a structure field, the structure is interpreted as by value**. To force the complementary interpretation, the tagging interfaces [Structure.ByValue](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByValue.html" \o "interface in com.sun.jna) and [Structure.ByReference](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByReference.html" \o "interface in com.sun.jna) are provided.

The data within a Java Structure is automatically written to native memory just before a native function call with a struct parameter, and automatically read from native memory after the function returns.

**Pointer-to-Structure Arguments**

To pass a pointer to a structure as an argument, simply use the Java structure subclass, and a pointer to native data memory will be used. The contents of the structure will be passed to the function and updated when the function returns. Structures are packed according to the default alignment rules for the platform's native C structs.

// Original C code

typedef struct \_Point {

int x, y;

} **Point**;

**Point\*** translate(**Point\*** pt, int dx, int dy);

// Equivalent JNA mapping

class **Point** extends Structure { public int x, y; }

Point translate(Point pt, int x, int y);

...

Point pt = new Point();

Point result = translate(pt, 100, 100);

**Structure by Value Arguments/Return**

To pass a structure by value, first define the structure, then define an empty class from that which implements [Structure.ByValue](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByValue.html" \o "interface in com.sun.jna). Use the ByValue class as the argument or return type.

// Original C code

typedef struct \_Point {

int x, y;

} Point;

Point translate(**Point** pt, int dx, int dy);

// Equivalent JNA mapping

class Point extends Structure {

public static class ByValue extends Point implements Structure.ByValue { }

public int x, y;

}

Point.ByValue translate(**Point.ByValue** pt, int x, int y);

...

Point.ByValue pt = new **Point.ByValue**();

Point result = translate(pt, 100, 100);

**Array-of-Structure Arguments**

To pass an array of structures, simply use a Java array of the desired structure type. If the array is uninitialized, it will be auto-initialized prior to the function call.

// Original C code

void get\_devices(**struct Device[]**, int size);

// Equivalent JNA mapping

int size = ...

Device[] devices = **new Device[size]**;

lib.get\_devices(devices, **devices.length**);

Alternatively, you can reallocate a single Structure instance into an array as follows:

Device dev = new Device();

// As an array of Structure

Structure[] structs = dev.toArray(size);

// As an array of Device

Device[] devices = (Device[])dev.toArray(size);

**Returning an Array of struct**

Declare the method as returning a [Structure](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html) of the appropriate type, then invoke [Structure.toArray(int)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html" \l "toArray-int-) to convert to an array of initialized structures of the appropriate size. Note that your [Structure](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html) class must have a no-args constructor, and you are responsible for freeing the returned memory if applicable in whatever way is appropriate for the called function.

// Original C code

struct Display\* get\_displays(int\* pcount);

void free\_displays(struct Display\* displays);

// Equivalent JNA mapping

Display get\_displays(IntByReference pcount);

void free\_displays(Display[] displays);

...

IntByReference pcount = new IntByReference();

Display d = lib.get\_displays(pcount);

Display[] displays = (Display[])d.toArray(pcount.getValue());

...

lib.free\_displays(displays);

**Nested Structure Definitions**

Nested structures are treated as consecutive memory (as opposed to pointers to structures). For example**:**

// Original C code

typedef struct \_Point {

int x, y;

} Point;

typedef struct \_Line {

**Point** start;

Point end;

} Line;

// Equivalent JNA mapping

class Point extends Structure {

public int x, y;

}

class Line extends Structure {

public **Point** start;

public Point end;

}

Explicit initialization of nested structures is not required; the objects will be created as needed and properly mapped to the parent structure's memory.

If you need a pointer to a structure within your structure, you can use the [Structure.ByReference](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.ByReference.html" \o "interface in com.sun.jna) tagging interface to indicate the field should be treated as a pointer instead of inlining the full structure**.**

// Original C code

typedef struct \_Line2 {

**Point\*** p1;

Point\* p2;

} Line2;

// Equivalent JNA mapping

class Point extends Structure {

public static class ByReference extends Point implements Structure.ByReference { }

public int x, y;

}

class Line2 extends Structure {

public **Point.ByReference** p1;

public Point.ByReference p2;

}

The more general case is just a pointer to memory. This allows you to define the field without necessarily defining the inner structure itself, similar to declaring a struct without defining it in C:

// Original C code

typedef struct \_Line2 {

**Point\*** p1;

Point\* p2;

} Line2;

// Equivalent JNA mapping

class Line2 extends Structure {

public Pointer p1;

public **Pointer** p2;

}

Line2 line2;

Point p1, p2;

...

line2.p1 = p1.getPointer();

line2.p2 = p2.getPointer();

**Nested arrays**

Structures with nested arrays require an explicit constructor to ensure the structure size is properly calculated.

typedef struct \_Buffer {

**char** buf1[32];

char buf2[1024];

} Buffer;

class Buffer extends Structure {

public **byte[]** buf1 = new byte[32];

public byte[] buf2 = new byte[1024];

}

Calculation of the native size of the structure is deferred until the structure is actually used.

**Variable-sized structures**

Structures with variable size, or with primitive array elements, for example:

// Original C code

typedef struct \_Header {

int flags;

int buf\_length;

char buffer[1];

} Header;

require a constructor which establishes the required size for the structure and initializes things appropriately. For example:

// Equivalent JNA mapping

class Header extends Structure {

public int flags;

public int buf\_length;

public byte[] buffer;

public Header(int bufferSize) {

buffer = new byte[bufferSize];

buf\_length = buffer.length;

allocateMemory();

}

}

**Volatile fields**

Normally, JNA will write the entire contents of a Structure prior to a function call and read back from native memory after the function call. Sometimes a structure field is not intended for client use, gets modified asynchronously by hardware, or otherwise is effectively read-only. If you expect any fields of the structure to be modified by any agent outside your Java program, you should mark the field volatile. This prevents JNA from automatically updating the native memory from the Java value. You can still force an update of the native memory from the Java value by calling [Structure.writeField(String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html" \l "writeField-java.lang.String-) for the field in question.

class Data extends com.sun.jna.Structure {

public volatile int refCount;

public int value;

}

...

Data data = new Data();

In the above example, the field refCount will only be written to native memory based on the Java value with a call todata.writeField("refCount"). To obtain the current state of native memory, call [Structure.read()](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html" \l "read--) (to update the entire structure) or [data.readField("refCount")](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html" \l "readField-java.lang.String-) (to update just the refCount field).

**Read-only fields**

If you want to absolutely prevent Java code from modifying a Structure's contents, you may mark its fields final. Structure reads can still overwrite the values based on native memory contents, but no Java code will be able to modify any of the fields.

class ReadOnly extends com.sun.jna.Structure {

// Do not initialize the field here, or the compiler will inline the value!

public final int refCount;

{

// Initialize fields here, to ensure the values are not inlined

refCount = -1;

read();

// refCount might now have a different value

}

}

...

ReadOnly ro = new ReadOnly();

// Will not compile!

ro.refCount = 0;

Make certain you attend to the following:

1. All final fields should be initialized in the constructor.
2. If you call Structure.read() from anywhere but the constructor, keep in mind that the compiler and/or hotspot will be assuming field values will not change across that function call.

***Unions***

Unions are a special type of Structure. Each declared field within the union overlays the same space in native memory. When writing a union to native memory, you *must* specify which field is to be written by supplying the desired field's class to the [Union.setType(java.lang.Class<?>)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Union.html" \l "setType-java.lang.Class-) method. On read, all non-pointer-based fields will be initialized from native memory. Structure, String, and WString members will *not* be initialized unless they are selected via [Union.setType(java.lang.Class<?>)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Union.html" \l "setType-java.lang.Class-).

***Obtaining "last" error***

If a function sets the system error property ([errno](http://www.opengroup.org/onlinepubs/009695399/functions/errno.html) or [GetLastError()](http://msdn.microsoft.com/en-us/library/ms679360(VS.85).aspx)), the error code will be thrown as a [LastErrorException](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/LastErrorException.html" \o "class in com.sun.jna) if you declare the exception in your JNA mapping. Alternatively, you can use [Native.getLastError()](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "getLastError--) to retrieve it. Throwing an exception is preferred since it has better performance.

***Arbitrary Java Object arguments/return values***

In some cases, such as invoking native VM functions directly, it is necessary to pass Java objects to the native methods. By default, JNA disallows using any Java object that is not explicitly supported unless it derives from [NativeMapped](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeMapped.html" \o "interface in com.sun.jna), because it is generally unnecessary to use such objects and usually signals a programmer error. To avoid errors flagging the use of Java objects, use the library load option [Library.OPTION\_ALLOW\_OBJECTS](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Library.html" \l "OPTION_ALLOW_OBJECTS) with Boolean.TRUE.

**Invocation Mapping**

Sometimes native functions exist only as C preprocessor macros or as inline functions. If you need to do more than simply change the name of the invoked function (which can be handled via [Function Mapping](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#function-mapping)), an [InvocationMapper](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/InvocationMapper.html" \o "interface in com.sun.jna) allows you to arbitrarily reconfigure the function invocation, including changing the method name and reordering, adding, or removing arguments. See the [InvocationMapper](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/InvocationMapper.html" \o "interface in com.sun.jna) documentation for details.

**Library Global Data**

The method [NativeLibrary.getGlobalVariableAddress(java.lang.String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/NativeLibrary.html#getGlobalVariableAddress-java.lang.String-) may be used to obtain the address of global variables as a [Pointer](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Pointer.html). Pointer methods may then be used to read or write the value as appropriate for the variable type.

**VM Crash Protection**

It is not uncommon when defining a new library and writing tests to encounter memory access errors which crash the VM. These are often caused by improper mappings or invalid arguments passed to the native library. **To generate Java errors instead of crashing the VM, call [Native.setProtected(true)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "setProtected-boolean-)**. Not all platforms support this protection; if not, the value of [Native.isProtected()](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Native.html" \l "isProtected--) will remain false.

NOTE: When protected mode is enabled, you should make use of the jsig library, if available (see [Signal Chaining](http://download.oracle.com/javase/6/docs/technotes/guides/vm/signal-chaining.html)) to avoid interfering with the JVM's use of signals. In short, set the environment variable LD\_PRELOAD (or LD\_PRELOAD\_64) to the path to libjsig.soin your JRE lib directory (usually ${java.home}/lib/${os.arch}/libjsig.so) before launching your Java application.

LD\_PRELOAD=/home/oracle/fmw11g/oracle\_pfrd/jdk/jre/lib/i386/libjsig.so

LD\_PRELOAD\_64=($ORACLE\_HOME)/jdk/jre/lib/amd64/libjsig.so

**Performance**

***Use direct mapping of methods***

Using [**direct mapping**](http://java-native-access.github.io/jna/5.2.0/javadoc/overview-summary.html#direct-mapping) of methods makes native calls more efficiently than does interface mapping. Direct mapping does not support varargs calls or arrays of Pointer, String, or WString as an argument or return value. For optimium results, use only primitive arguments and return values; you'll have to convert to and from objects yourself explicitly.

***Avoid type mapping***

Type mapping incurs additional overhead on each function call. You can avoid this by ensuring that your arguments and/or return types are already primitive types.

***Pointer/Array/Buffer Variants***

Java primitive arrays are generally slower to use than direct memory (Pointer, Memory, or ByReference) or NIO buffers, since the Java memory has to be pinned and possibly copied across the native call, since the **Java array is not necessarily contiguously allocated**.

***Large Structures***

Structures are normally written to native memory before and read back from native memory after a function call. With very large structures, there can be a performance hit using reflection to walk through all the fields. Structure auto-synch can be disabled by calling [Structure.setAutoSynch(boolean)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html" \l "setAutoSynch-boolean-) with a false parameter. It is then up to you to use [Structure.readField(String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html#readField-java.lang.String-) and [Structure.writeField(String)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html#writeField-java.lang.String-) or [Structure.writeField(String,Object)](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/Structure.html#writeField-java.lang.String-java.lang.Object-) to synch with just the fields of interest.

***Throw exceptions on last error***

In those methods where you are interested in the value of errno/GetLastError(), declare your method to throw [LastErrorException](http://java-native-access.github.io/jna/5.2.0/javadoc/com/sun/jna/LastErrorException.html" \o "class in com.sun.jna)**.**

**JNA API 5.2.0**

***Copyright © 2007-2018 Timothy Wall. All Rights Reserved.***