Java Native Interface Specification

Release 1.1 (Revised May, 1997)

JAVASOFT
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043 U.S.A.
408-343-1400

May 1997

Copyright Information

© 1996, 1997 Sun Microsystems, Inc. All rights reserved.
2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A.

This document is protected by copyright. No part of this document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

The information described in this document may be protected by one or more U.S. patents, foreign patents, or pending applications.

TRADEMARKS

Sun, Sun Microsystems, Sun Microelectronics, the Sun Logo, SunXTL, JavaSoft, JavaOS, the JavaSoft Logo, Java, HotJava, JavaChips, picoJava, microJava, UltraJava, JDBC, the Java Cup and Steam Logo, "Write Once, Run Anywhere" and Solaris are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.
UNIX® is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.
Adobe® is a registered trademark of Adobe Systems, Inc.
Netscape Navigator™ is a trademark of Netscape Communications Corporation.
All other product names mentioned herein are the trademarks of their respective owners.

THIS DOCUMENT IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS DOCUMENT COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE DOCUMENT. SUN MICROSYSTEMS, INC. MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS DOCUMENT AT ANY TIME.

Please
Recycle

Contents

Acknowledgments

The Java Native Interface (JNI) specification was formed as a result of a series of discussions among JavaSoft and Java licensees. The goal is to achieve a native interface standard.

Sheng Liang at JavaSoft is responsible for the JNI design and specification. JNI is partly evolved from Netscape's Java Runtime Interface (JRI), which was designed by Warren Harris. Warren Harris also helped to improve the JNI design. Simon Nash at IBM and Patrick Beard at Apple provided extensive feedbacks that shaped the JNI design in many ways.

In addition, the JNI benefited greatly from JavaSoft internal design reviews. These design reviews were conducted by James Gosling, Peter Kessler, Tim Lindholm, Mark Reinhold, Derek White, and Frank Yellin.

We also want to acknowledge the helpful comments and suggestions we received from numerous people on various drafts of the specification.

Lastly, Beth Stearns improved the presentation of this document.

Introduction 1

This chapter introduces the *Java Native Interface* (JNI). The JNI is a native programming interface. It allows Java code that runs inside a Java Virtual Machine (VM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly.

The most important benefit of the JNI is that it imposes no restrictions on the implementation of the underlying Java VM. Therefore, Java VM vendors can add support for the JNI without affecting other parts of the VM. Programmers can write one version of a native application or library and expect it to work with all Java VMs supporting the JNI.

This chapter covers the following topics:

- *Java Native Interface Overview*
- *Background*
- *Objectives*
- *Java Native Interface Approach*
- *Programming to the JNI*
- *Changes in JDK 1.1.2*

Java Native Interface Overview

While you can write applications entirely in Java, there are situations where Java alone does not meet the needs of your application. Programmers use the JNI to write *Java native methods* to handle those situations when an application cannot be written entirely in Java.

The following examples illustrate when you need to use Java native methods:

- The standard Java class library does not support the platform-dependent features needed by the application.
- You already have a library written in another language, and wish to make it accessible to Java code through the JNI.
- You want to implement a small portion of time-critical code in a lower-level language such as assembly.

By programming through the JNI, you can use native methods to:

- Create, inspect, and update Java objects (including arrays and strings).
- Call Java methods.
- Catch and throw exceptions.
- Load classes and obtain class information.
- Perform runtime type checking.

You can also use the JNI with the *Invocation API* to enable an arbitrary native application to embed the Java VM. This allows programmers to easily make their existing applications Java-enabled without having to link with the VM source code.

Background

Currently, VMs from different vendors offer different native method interfaces. These different interfaces force programmers to produce, maintain, and distribute multiple versions of native method libraries on a given platform.

We briefly examine some of the existing native method interfaces, such as:

- JDK 1.0 native method interface
- Netscape's Java Runtime Interface
- Microsoft's Raw Native Interface and Java/COM interface

JDK 1.0 Native Method Interface

JDK 1.0 shipped with a native method interface. Unfortunately, there are two major reasons that this interface is unsuitable for adoption by other Java VMs.

First, the native code accesses fields in Java objects as members of C structures. However, the *Java Language Specification* does not define how objects are laid out in memory. If a Java VM lays out objects differently in memory, then the programmer would have to recompile the native method libraries.

Second, JDK 1.0's native method interface relies on a conservative garbage collector. The unrestricted use of the `unhand` macro, for example, makes it necessary to conservatively scan the native stack.

Java Runtime Interface

Netscape proposed the Java Runtime Interface (JRI), a general interface for services provided in the Java virtual machine. JRI is designed with portability in mind---it makes few assumptions about the implementation details in the underlying Java VM. The JRI addresses a wide range of issues, including native methods, debugging, reflection, embedding (invocation), and so on.

Raw Native Interface and Java/COM Interface

The Microsoft Java VM supports two native method interfaces. At the low level, it provides an efficient Raw Native Interface (RNI). The RNI offers a high degree of source-level backward compatibility with the JDK's native method interface, although it has one major difference. Instead of relying on conservative garbage collection, the native code must use RNI functions to interact explicitly with the garbage collector.

At a higher level, Microsoft's Java/COM interface offers a language-independent standard binary interface to the Java VM. Java code can use a COM object as if it were a Java object. A Java class can also be exposed to the rest of the system as a COM class.

Objectives

We believe that a uniform, well-thought-out standard interface offers the following benefits for everyone:

- Each VM vendor can support a larger body of native code.
- Tool builders will not have to maintain different kinds of native method interfaces.

- Application programmers will be able to write one version of their native code and this version will run on different VMs.

The best way to achieve a standard native method interface is to involve all parties with an interest in Java VMs. Therefore we organized a series of discussions among the Java licensees on the design of a uniform native method interface. It is clear from the discussions that the standard native method interface must satisfy the following requirements:

- Binary compatibility - The primary goal is binary compatibility of native method libraries across all Java VM implementations on a given platform. Programmers should maintain only one version of their native method libraries for a given platform.
- Efficiency - To support time-critical code, the native method interface must impose little overhead. All known techniques to ensure VM-independence (and thus binary compatibility) carry a certain amount of overhead. We must somehow strike a compromise between efficiency and VM-independence.
- Functionality - The interface must expose enough Java VM internals to allow native methods to accomplish useful tasks.

Java Native Interface Approach

We hoped to adopt one of the existing approaches as the standard interface, because this would have imposed the least burden on programmers who had to learn multiple interfaces in different VMs. Unfortunately, no existing solutions are completely satisfactory in achieving our goals.

Netscape's JRI is the closest to what we envision as a portable native method interface, and was used as the starting point of our design. Readers familiar with the JRI will notice the similarities in the API naming convention, the use of method and field IDs, the use of local and global references, and so on. Despite our best efforts, however, the JNI is not binary-compatible with the JRI, although a VM can support both the JRI and the JNI.

Microsoft's RNI is an improvement over JDK 1.0 because it solves the problem of native methods working with a nonconservative garbage collector. The RNI, however, is not suitable as a VM-independent native method interface. Like the JDK, RNI native methods access Java objects as C structures. This leads to two problems:

- RNI exposes the layout of internal Java objects to native code.

- Direct access of Java objects as C structures makes it impossible to efficiently incorporate "write barriers," which are necessary in advanced garbage collection algorithms.

As a binary standard, COM ensures complete binary compatibility across different VMs. Invoking a COM method requires only an indirect call, which carries little overhead. In addition, COM objects are a great improvement over dynamic-link libraries in solving versioning problems.

The use of COM as the standard Java native method interface, however, is hampered by a few factors:

- First, the Java/COM interface lacks certain desired functions, such as accessing private fields and raising general exceptions.
- Second, the Java/COM interface automatically provides the standard `IUnknown` and `IDispatch` COM interfaces for Java objects, so that native code can access public methods and fields. Unfortunately, the `IDispatch` interface does not deal with overloaded Java methods and is case-insensitive in matching method names. Furthermore, all Java methods exposed through the `IDispatch` interface are wrapped to perform dynamic type checking and coercion. This is because the `IDispatch` interface is designed with weakly-typed languages (such as Basic) in mind.
- Third, instead of dealing with individual low-level functions, COM is designed to allow software components (including full-fledged applications) to work together. We believe that it is not appropriate to treat all Java classes or low-level native methods as software components.
- Fourth, the immediate adoption of COM is hampered by the lack of its support on UNIX platforms.

Although we do not expose Java objects to the native code as COM objects, the JNI interface itself is binary-compatible with COM. We use the same jump table structure and calling convention that COM does. *This means that, as soon as cross-platform support for COM is available, the JNI can become a COM interface to the Java VM.*

We do not believe that the JNI should be the only native method interface supported by a given Java VM. A standard interface benefits programmers who would like to load their native code libraries into different Java VMs. In some cases, the programmer may have to use a lower-level, VM-specific interface to achieve top efficiency. In other cases, the programmer might use a higher-level interface to build software components. Indeed, we hope that, as the Java environment and component software technologies become more mature, native methods will gradually lose their significance.

Programming to the JNI

Native method programmers should start programming to the JNI. Programming to the JNI insulates you from unknowns, such as the vendor's VM that the end user might be running. By conforming to the JNI standard, you will give a native library the best chance to run in a given Java VM. For example, although JDK 1.1 will continue to support the old-style native method interface that was implemented in JDK 1.0, it is certain that future versions of the JDK will stop supporting the old-style native method interface. Native methods relying on the old-style interface will have to be rewritten.

If you are implementing a Java VM, you should implement the JNI. We (Javasoft and the licensees) have tried our best to ensure that the JNI does not impose any overhead or restrictions on your VM implementation, including object representation, garbage collection scheme, and so on. Please let us know if you run into any problems we might have overlooked.

Changes in JDK 1.1.2

To better support the Java Runtime Environment (JRE), the Invocation API has been extended in JDK 1.1.2 in a few minor ways. The changes do not break any existing code. The JNI Native Method Interface has not been changed.

- The `reserved0` field in the JDK1_1InitArgs structure has been renamed to `version`. The `JDK1_1InitArgs` structure holds the initialization arguments to `JNI_CreateJavaVM`. Callers of `JNI_GetDefaultJavaVMInitArgs` and `JNI_CreateJavaVM` must set the version field to `0x00010001`. `JNI_GetDefaultJavaVMInitArgs` has been changed to return a `jint` indicating whether the requested version is supported.
- The `reserved1` field in the `JDK1_1InitArgs` structure has been renamed to `properties`. This is a `NULL`-terminated array of strings. Each string has the format:

  name=value

  indicating a system property. (This facility corresponds to the -D option in the java command line.)

- In JDK 1.1.1, the thread calling `DestroyJavaVM` must be the only user thread in the VM. JDK 1.1.2 has lifted this restriction. If `DestroyJavaVM` is called when there is more than one user thread, the VM waits until the current thread is the only user thread, and then tries to destroy itself.

Design Overview

2

This chapter focuses on major design issues in the JNI. Most design issues in this section are related to native methods. The design of the Invocation API is covered in Chapter 5, “The Invocation API.”

JNI Interface Functions and Pointers

Native code accesses Java VM features by calling JNI functions. JNI functions are available through an *interface pointer*. An interface pointer is a pointer to a pointer. This pointer points to an array of pointers, each of which points to an interface function. Every interface function is at a predefined offset inside the array. Figure 2-1 illustrates the organization of an interface pointer.

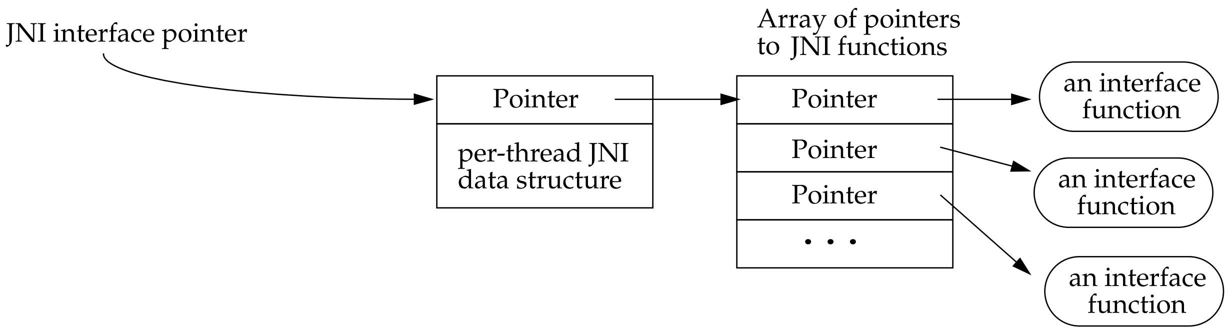

Figure 2-1 Interface Pointer

The JNI interface is organized like a C++ virtual function table or a COM interface. The advantage to using an interface table, rather than hard-wired function entries, is that the JNI name space becomes separate from the native code. A VM can easily provide multiple versions of JNI function tables. For example, the VM may support two JNI function tables:

- one performs thorough illegal argument checks, and is suitable for debugging;
- the other performs the minimal amount of checking required by the JNI specification, and is therefore more efficient.

The JNI interface pointer is only valid in the current thread. A native method, therefore, must not pass the interface pointer from one thread to another. A VM implementing the JNI may allocate and store thread-local data in the area pointed to by the JNI interface pointer.

Native methods receive the JNI interface pointer as an argument. The VM is guaranteed to pass the same interface pointer to a native method when it makes multiple calls to the native method from the same Java thread. However, a native method can be called from different Java threads, and therefore may receive different JNI interface pointers.

Loading and Linking Native Methods

Native methods are loaded with the `System.loadLibrary` method. In the following example, the class initialization method loads a platform-specific native library in which the native method `f` is defined:

```
package pkg;
class Cls {
     native double f(int i, String s);
     static {
         System.loadLibrary("pkg_Cls");
     }
}
```

The argument to `System.loadLibrary` is a library name chosen arbitrarily by the programmer. The system follows a standard, but platform-specific, approach to convert the library name to a native library name. For example, a Solaris system converts the name `pkg_Cls` to `libpkg_Cls.so`, while a Win32 system converts the same `pkg_Cls` name to `pkg_Cls.dll`.

The programmer may use a single library to store all the native methods needed by any number of classes, as long as these classes are to be loaded with the same class loader. The VM internally maintains a list of loaded native libraries for each class loader. Vendors should choose native library names that minimize the chance of name clashes.

If the underlying operating system does not support dynamic linking, all native methods must be prelinked with the VM. In this case, the VM completes the `System.loadLibrary` call without actually loading the library.

The programmer can also call the JNI function `RegisterNatives()` to register the native methods associated with a class. The `RegisterNatives()` function is particularly useful with statically linked functions.

Resolving Native Method Names

Dynamic linkers resolve entries based on their names. A native method name is concatenated from the following components:

- the prefix `Java_`
- a mangled fully-qualified class name
- an underscore ("_") separator
- a mangled method name
- for overloaded native methods, two underscores ("__") followed by the mangled argument signature

The VM checks for a method name match for methods that reside in the native library. The VM looks first for the short name; that is, the name without the argument signature. It then looks for the long name, which is the name with the argument signature. Programmers need to use the long name only when a native method is overloaded with another native method. However, this is not a problem if the native method has the same name as a nonnative method. A nonnative method (a Java method) does not reside in the native library.

In the following example, the native method `g` does not have to be linked using the long name because the other method `g` is not a native method, and thus is not in the native library.

```
class Cls1 {
  int g(int i);
  native int g(double d);
}
```

We adopted a simple name-mangling scheme to ensure that all Unicode characters translate into valid C function names. We use the underscore ("_")

character as the substitute for the slash ("/") in fully qualified class names. Since a name or type descriptor never begins with a number, we can use `_0`, ..., `_9` for escape sequences, as Table 2-1 illustrates:

Table 2-1 Unicode Character Translation

Escape Sequence	**Denotes**
`_0XXXX`	a Unicode character `XXXX`.
`_1`	the character "_"
`_2`	the character ";" in signatures
`_3`	the character "[" in signatures

Both the native methods and the interface APIs follow the standard library-calling convention on a given platform. For example, UNIX systems use the C calling convention, while Win32 systems use `__stdcall`.

Native Method Arguments

The JNI interface pointer is the first argument to native methods. The JNI interface pointer is of type *JNIEnv*. The second argument differs depending on whether the native method is static or nonstatic. The second argument to a nonstatic native method is a reference to the object. The second argument to a static native method is a reference to its Java class.

The remaining arguments correspond to regular Java method arguments. The native method call passes its result back to the calling routine via the return value. Chapter 3, "JNI Types and Data Structures," describes the mapping between Java and C types.

Code Example 2-1 illustrates using a C function to implement the native method `f`. The native method `f` is declared as follows:

```
package pkg;
class Cls {
     native double f(int i, String s);
     ...
}
```

The C function with the long mangled name `Java_pkg_Cls_f_ILjava_lang_String_2` implements native method `f`:

Code Example 2-1 Implementing a Native Method Using C

```
jdouble Java_pkg_Cls_f__ILjava_lang_String_2 (
     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
     jint i,             /* argument #1 */
     jstring s)          /* argument #2 */
{
     /* Obtain a C-copy of the Java string */
     const char *str = (*env)->GetStringUTFChars(env, s, 0);
     /* process the string */
     ...
     /* Now we are done with str */
     (*env)->ReleaseStringUTFChars(env, s, str);
     return ...
}
```

Note that we always manipulate Java objects using the interface pointer `env` . Using C++, you can write a slightly cleaner version of the code, as shown in Code Example 2-2:

Code Example 2-2 Implementing a Native Method Using C++

```
extern "C" /* specify the C calling convention */
jdouble Java_pkg_Cls_f__ILjava_lang_String_2 (
     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
     jint i,             /* argument #1 */
     jstring s)          /* argument #2 */
{
     const char *str = env->GetStringUTFChars(s, 0);
     ...
     env->ReleaseStringUTFChars(s, str);
     return ...
}
```

With C++, the extra level of indirection and the interface pointer argument disappear from the source code. However, the underlying mechanism is exactly the same as with C. In C++, JNI functions are defined as inline member functions that expand to their C counterparts.

Referencing Java Objects

Primitive types, such as integers, characters, and so on, are copied between Java and native code. Arbitrary Java objects, on the other hand, are passed by reference. The VM must keep track of all objects that have been passed to the native code, so that these objects are not freed by the garbage collector. The native code, in turn, must have a way to inform the VM that it no longer needs the objects. In addition, the garbage collector must be able to move an object referred to by the native code.

Global and Local References

The JNI divides object references used by the native code into two categories: *local* and *global references*. Local references are valid for the duration of a native method call, and are automatically freed after the native method returns. Global references remain valid until they are explicitly freed.

Objects are passed to native methods as local references. All Java objects returned by JNI functions are local references. The JNI allows the programmer to create global references from local references. JNI functions that expect Java objects accept both global and local references. A native method may return a local or global reference to the VM as its result.

In most cases, the programmer should rely on the VM to free all local references after the native method returns. However, there are times when the programmer should explicitly free a local reference. Consider, for example, the following situations:

- A native method accesses a large Java object, thereby creating a local reference to the Java object. The native method then performs additional computation before returning to the caller. The local reference to the large Java object will prevent the object from being garbage collected, even if the object is no longer used in the remainder of the computation.
- A native method creates a large number of local references, although not all of them are used at the same time. Since the VM needs a certain amount of space to keep track of a local reference, creating too many local references may cause the system to run out of memory. For example, a native method loops through a large array of objects, retrieves the elements as local references, and operates on one element at each iteration. After each iteration, the programmer no longer needs the local reference to the array element.

The JNI allows the programmer to manually delete local references at any point within a native method. To ensure that programmers can manually free local references, JNI functions are not allowed to create extra local references, except for references they return as the result.

Local references are only valid in the thread in which they are created. The native code must not pass local references from one thread to another.

Implementing Local References

To implement local references, the Java VM creates a registry for each transition of control from Java to a native method. A registry maps nonmovable local references to Java objects, and keeps the objects from being garbage collected. All Java objects passed to the native method (including those that are returned as the results of JNI function calls) are automatically added to the registry. The registry is deleted after the native method returns, allowing all of its entries to be garbage collected.

There are different ways to implement a registry, such as using a table, a linked list, or a hash table. Although reference counting may be used to avoid duplicated entries in the registry, a JNI implementation is not obliged to detect and collapse duplicate entries.

Note that local references cannot be faithfully implemented by conservatively scanning the native stack. The native code may store local references into global or heap data structures.

Accessing Java Objects

The JNI provides a rich set of accessor functions on global and local references. This means that the same native method implementation works no matter how the VM represents Java objects internally. This is a crucial reason why the JNI can be supported by a wide variety of VM implementations.

The overhead of using accessor functions through opaque references is higher than that of direct access to C data structures. We believe that, in most cases, Java programmers use native methods to perform nontrivial tasks that overshadow the overhead of this interface.

Accessing Primitive Arrays

This overhead is not acceptable for large Java objects containing many primitive data types, such as integer arrays and strings. (Consider native methods that are used to perform vector and matrix calculations.) It would be grossly inefficient to iterate through a Java array and retrieve every element with a function call.

One solution introduces a notion of "pinning" so that the native method can ask the VM to pin down the contents of an array. The native method then receives a direct pointer to the elements. This approach, however, has two implications:

- The garbage collector must support pinning.
- The VM must lay out primitive arrays contiguously in memory. Although this is the most natural implementation for most primitive arrays, boolean arrays can be implemented as packed or unpacked. Therefore, native code that relies on the exact layout of boolean arrays will not be portable.

We adopt a compromise that overcomes both of the above problems.

First, we provide a set of functions to copy primitive array elements between a segment of a Java array and a native memory buffer. Use these functions if a native method needs access to only a small number of elements in a large array.

Second, programmers can use another set of functions to retrieve a pinned-down version of array elements. Keep in mind that these functions may require the Java VM to perform storage allocation and copying. Whether these functions in fact copy the array depends on the VM implementation, as follows:

- If the garbage collector supports pinning, and the layout of the array is the same as expected by the native method, then no copying is needed.
- Otherwise, the array is copied to a nonmovable memory block (for example, in the C heap) and the necessary format conversion is performed. A pointer to the copy is returned.

Lastly, the interface provides functions to inform the VM that the native code no longer needs to access the array elements. When you call these functions, the system either unpins the array, or it reconciles the original array with its non-movable copy and frees the copy.

Our approach provides flexibility. A garbage collector algorithm can make separate decisions about copying or pinning for each given array. For example, the garbage collector may copy small objects, but pin the larger objects.

A JNI implementation must ensure that native methods running in multiple threads can simultaneously access the same array. For example, the JNI may keep an internal counter for each pinned array so that one thread does not unpin an array that is also pinned by another thread. Note that the JNI does not need to lock primitive arrays for exclusive access by a native method. Simultaneously updating a Java array from different threads leads to nondeterministic results.

Accessing Fields and Methods

The JNI allows native code to access the fields and to call the methods of Java objects. The JNI identifies methods and fields by their symbolic names and type signatures. A two-step process factors out the cost of locating the field or method from its name and signature. For example, to call the method `f` in class `cls`, the native code first obtains a method ID, as follows:

```
jmethodID mid =
     env->GetMethodID(cls, “f”, “(ILjava/lang/String;)D”);
```

The native code can then use the method ID repeatedly without the cost of method lookup, as follows:

```
jdouble result = env->CallDoubleMethod(obj, mid, 10, str);
```

A field or method ID does not prevent the VM from unloading the class from which the ID has been derived. After the class is unloaded, the method or field ID becomes invalid. The native code, therefore, must make sure to:

- keep a live reference to the underlying class, or
- recompute the method or field ID

if it intends to use a method or field ID for an extended period of time.

The JNI does not impose any restrictions on how field and method IDs are implemented internally.

Reporting Programming Errors

The JNI does not check for programming errors such as passing in `NULL` pointers or illegal argument types. Illegal argument types includes such things as using a normal Java object instead of a Java class object. The JNI does not check for these programming errors for the following reasons:

- Forcing JNI functions to check for all possible error conditions degrades the performance of normal (correct) native methods.
- In many cases, there is not enough runtime type information to perform such checking.

Most C library functions do not guard against programming errors. The `printf()` function, for example, usually causes a runtime error, rather than returning an error code, when it receives an invalid address. Forcing C library functions to check for all possible error conditions would likely result in such checks to be duplicated--once in the user code, and then again in the library.

The programmer must not pass illegal pointers or arguments of the wrong type to JNI functions. Doing so could result in arbitrary consequences, including a corrupted system state or VM crash.

Java Exceptions

The JNI allows native methods to raise arbitrary Java exceptions. The native code may also handle outstanding Java exceptions. The Java exceptions left unhandled are propagated back to the VM.

Exceptions and Error Codes

Certain JNI functions use the Java exception mechanism to report error conditions. In most cases, JNI functions report error conditions by returning an error code *and* throwing a Java exception. The error code is usually a special return value (such as `NULL`) that is outside of the range of normal return values. Therefore, the programmer can:

- quickly check the return value of the last JNI call to determine if an error has occurred, and
- call a function, `ExceptionOccurred()`, to obtain the exception object that contains a more detailed description of the error condition.

There are two cases where the programmer needs to check for exceptions without being able to first check an error code:

- The JNI functions that invoke a Java method return the result of the Java method. The programmer must call `ExceptionOccurred()` to check for possible exceptions that occurred during the execution of the Java method.
- Some of the JNI array access functions do not return an error code, but may throw an `ArrayIndexOutOfBoundsException` or `ArrayStoreException`.

In all other cases, a non-error return value guarantees that no exceptions have been thrown.

Asynchronous Exceptions

In cases of multiple threads, threads other than the current thread may post an asynchronous exception. An asynchronous exception does not immediately affect the execution of the native code in the current thread, until:

- the native code calls one of the JNI functions that could raise synchronous exceptions, or
- the native code uses `ExceptionOccurred()` to explicitly check for synchronous and asynchronous exceptions.

Note that only those JNI function that could potentially raise synchronous exceptions check for asynchronous exceptions.

Native methods should insert `ExceptionOccurred()` checks in necessary places (such as in a tight loop without other exception checks) to ensure that the current thread responds to asynchronous exceptions in a reasonable amount of time.

Exception Handling

There are two ways to handle an exception in native code:

- The native method can choose to return immediately, causing the exception to be thrown in the Java code that initiated the native method call.
- The native code can clear the exception by calling `ExceptionClear()`, and then execute its own exception-handling code.

After an exception has been raised, the native code must first clear the exception before making other JNI calls. When there is a pending exception, the only JNI functions that are safe to call are `ExceptionOccurred()`, `ExceptionDescribe()`, and `ExceptionClear()`. The

`ExceptionDescribe()` function prints a debugging message about the pending exception.

JNI Types and Data Structures 3

This chapter discusses how the JNI maps Java types to native C types.

Primitive Types

Table 3-1 describes Java primitive types and their machine-dependent native equivalents.

Table 3-1 Primitive Types and Native Equivalents

Java Type	Native Type	Description
`boolean`	`jboolean`	unsigned 8 bits
`byte`	`jbyte`	signed 8 bits
`char`	`jchar`	unsigned 16 bits
`short`	`jshort`	signed 16 bits
`int`	`jint`	signed 32 bits
`long`	`jlong`	signed 64 bits
`float`	`jfloat`	32 bits
`double`	`jdouble`	64 bits
`void`	`void`	N/A

The following definition is provided for convenience.

```
#define JNI_FALSE  0
#define JNI_TRUE   1
```

The `jsize` integer type is used to describe cardinal indices and sizes:

```
typedef jint jsize;
```

Reference Types

The JNI includes a number of reference types that correspond to different kinds of Java objects. JNI reference types are organized in the hierarchy shown in Figure 3-1.

```
jobject                          (all Java objects)
    jclass                       (java.lang.Class objects)
    jstring                      (java.lang.String objects)
    jarray                       (arrays)
        jobjectArray             (object arrays)
        jbooleanArray            (boolean arrays)
        jbyteArray               (byte arrays)
        jcharArray               (char arrays)
        jshortArray              (short arrays)
        jintArray                (int arrays)
        jlongArray               (long arrays)
        jfloatArray              (float arrays)
        jdoubleArray             (double arrays)
    jthrowable                   (java.lang.Throwable objects)
```

Figure 3-1 Reference Type Hierarchy

In C, all other JNI reference types are defined to be the same as `jobject`. For example:

```
typedef jobject jclass;
```

In C++, JNI introduces a set of dummy classes to enforce the subtyping relationship. For example:

```
class _jobject {};
class _jclass : public _jobject {};
...
typedef _jobject *jobject;
typedef _jclass *jclass;
```

Field and Method IDs

Method and field IDs are regular C pointer types:

```
struct _jfieldID;              /* opaque structure */
typedef struct _jfieldID *jfieldID;   /* field IDs */

struct _jmethodID;              /* opaque structure */
typedef struct _jmethodID *jmethodID; /* method IDs */
```

The Value Type

The `jvalue` union type is used as the element type in argument arrays. It is declared as follows:

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

Type Signatures

The JNI uses the Java VM's representation of type signatures. Table 3-2 shows these type signatures.

Table 3-2 Java VM Type Signatures

Type Signature	Java Type
Z	`boolean`
B	`byte`
C	`char`
S	`short`

Table 3-2 Java VM Type Signatures

Type Signature	Java Type
I	`int`
J	`long`
F	`float`
D	`double`
L *fully-qualified-class* **;**	*fully-qualified-class*
[*type*	*type* `[]`
(*arg-types* **)** *ret-type*	method type

For example, the Java method:

```
long f (int n, String s, int[] arr);
```

has the following type signature:

```
(ILjava/lang/String;[I)J
```

UTF-8 Strings

The JNI uses UTF-8 strings to represent various string types. UTF-8 strings are the same as those used by the Java VM. UTF-8 strings are encoded so that character sequences that contain only nonnull ASCII characters can be represented using only one byte per character, but characters of up to 16 bits can be represented. All characters in the range `\u0001` to `\u007F` are represented by a single byte, as follows:

0	bits 0-6

The seven bits of data in the byte give the value of the character that is represented. The null character (`\u000`) and characters in the range `\u0080 to \u07FF` are represented by a pair of bytes, *x* and *y*, as follows:

`x:`	1	1	0	bits 6-10	`y:`	1	0	bits 0-5

The bytes represent the character with the value `((x&0x1f)<<6)+(y&0x3f)`.

Characters in the range `\u0800` to `\uFFFF` are represented by three bytes, *x*, *y*, and *z*:

x:	1	1	1	0	bits 12-15	y:	1	0	bits 6-11	z:	1	0	bits 0-5

The character with the value `((x&0xf)<<12)+(y&0x3f)<<6)+(z&0x3f)` is represented by the three bytes.

There are two differences between this format and the "standard" UTF-8 format. First, the null byte `(byte)0` is encoded using the two-byte format rather than the one-byte format. This means that Java VM UTF-8 strings never have embedded nulls. Second, only the one-byte, two-byte, and three-byte formats are used. The Java VM does not recognize the longer UTF-8 formats.

JNI Functions 4

This chapter serves as the reference section for the JNI functions. It provides a complete listing of all the JNI functions. It also presents the exact layout of the JNI function table.

Note the use of the term "must" to describe restrictions on JNI programmers. For example, when you see that a certain JNI function *must* receive a non-`NULL` object, it is your responsibility to ensure that `NULL` is not passed to that JNI function. As a result, a JNI implementation does not need to perform `NULL` pointer checks in that JNI function.

A portion of this chapter is adapted from Netscape's JRI documentation.

The reference material groups functions by their usage. The reference section is organized by the following functional areas:

- *Version Information*
- *Class Operations*
- *Exceptions*
- *Global and Local References*
- *Object Operations*
- *Accessing Fields of Objects*
- *Calling Instance Methods*
- *Accessing Static Fields*
- *Calling Static Methods*
- *String Operations*

- *Array Operations*
- *Registering Native Methods*
- *Monitor Operations*
- *Java VM Interface*

Interface Function Table

Each function is accessible at a fixed offset through the *JNIEnv* argument. The *JNIEnv* type is a pointer to a structure storing all JNI function pointers. It is defined as follows:

```
typedef const struct JNINativeInterface *JNIEnv;
```

The VM initializes the function table, as shown by Code Example 4-1. Note that the first three entries are reserved for future compatibility with COM. In addition, we reserve a number of additional `NULL` entries near the beginning of the function table, so that, for example, a future class-related JNI operation can be added after `FindClass`, rather than at the end of the table.

Note that the function table can be shared among all JNI interface pointers.

Code Example 4-1

```
const struct JNINativeInterface ... = {
    NULL,
    NULL,
    NULL,
    NULL,
    GetVersion,

    DefineClass,
    FindClass,
    NULL,
    NULL,
    NULL,
    GetSuperclass,
    IsAssignableFrom,
    NULL,

    Throw,
    ThrowNew,
    ExceptionOccurred,
```

Code Example 4-1

```
    ExceptionDescribe,
    ExceptionClear,
    FatalError,
    NULL,
    NULL,

    NewGlobalRef,
    DeleteGlobalRef,
    DeleteLocalRef,
    IsSameObject,
    NULL,
    NULL,

    AllocObject,
    NewObject,
    NewObjectV,
    NewObjectA,

    GetObjectClass,
    IsInstanceOf,

    GetMethodID,

    CallObjectMethod,
    CallObjectMethodV,
    CallObjectMethodA,
    CallBooleanMethod,
    CallBooleanMethodV,
    CallBooleanMethodA,
    CallByteMethod,
    CallByteMethodV,
    CallByteMethodA,
    CallCharMethod,
    CallCharMethodV,
    CallCharMethodA,
    CallShortMethod,
    CallShortMethodV,
    CallShortMethodA,
    CallIntMethod,
    CallIntMethodV,
    CallIntMethodA,
    CallLongMethod,
    CallLongMethodV,
    CallLongMethodA,
```

Code Example 4-1

```
    CallFloatMethod,
    CallFloatMethodV,
    CallFloatMethodA,
    CallDoubleMethod,
    CallDoubleMethodV,
    CallDoubleMethodA,
    CallVoidMethod,
    CallVoidMethodV,
    CallVoidMethodA,

    CallNonvirtualObjectMethod,
    CallNonvirtualObjectMethodV,
    CallNonvirtualObjectMethodA,
    CallNonvirtualBooleanMethod,
    CallNonvirtualBooleanMethodV,
    CallNonvirtualBooleanMethodA,
    CallNonvirtualByteMethod,
    CallNonvirtualByteMethodV,
    CallNonvirtualByteMethodA,
    CallNonvirtualCharMethod,
    CallNonvirtualCharMethodV,
    CallNonvirtualCharMethodA,
    CallNonvirtualShortMethod,
    CallNonvirtualShortMethodV,
    CallNonvirtualShortMethodA,
    CallNonvirtualIntMethod,
    CallNonvirtualIntMethodV,
    CallNonvirtualIntMethodA,
    CallNonvirtualLongMethod,
    CallNonvirtualLongMethodV,
    CallNonvirtualLongMethodA,
    CallNonvirtualFloatMethod,
    CallNonvirtualFloatMethodV,
    CallNonvirtualFloatMethodA,
    CallNonvirtualDoubleMethod,
    CallNonvirtualDoubleMethodV,
    CallNonvirtualDoubleMethodA,
    CallNonvirtualVoidMethod,
    CallNonvirtualVoidMethodV,
    CallNonvirtualVoidMethodA,

    GetFieldID,

    GetObjectField,
```

Code Example 4-1

```
    GetBooleanField,
    GetByteField,
    GetCharField,
    GetShortField,
    GetIntField,
    GetLongField,
    GetFloatField,
    GetDoubleField,
    SetObjectField,
    SetBooleanField,
    SetByteField,
    SetCharField,
    SetShortField,
    SetIntField,
    SetLongField,
    SetFloatField,
    SetDoubleField,

    GetStaticMethodID,

    CallStaticObjectMethod,
    CallStaticObjectMethodV,
    CallStaticObjectMethodA,
    CallStaticBooleanMethod,
    CallStaticBooleanMethodV,
    CallStaticBooleanMethodA,
    CallStaticByteMethod,
    CallStaticByteMethodV,
    CallStaticByteMethodA,
    CallStaticCharMethod,
    CallStaticCharMethodV,
    CallStaticCharMethodA,
    CallStaticShortMethod,
    CallStaticShortMethodV,
    CallStaticShortMethodA,
    CallStaticIntMethod,
    CallStaticIntMethodV,
    CallStaticIntMethodA,
    CallStaticLongMethod,
    CallStaticLongMethodV,
    CallStaticLongMethodA,
    CallStaticFloatMethod,
    CallStaticFloatMethodV,
    CallStaticFloatMethodA,
```

Code Example 4-1

```
    CallStaticDoubleMethod,
    CallStaticDoubleMethodV,
    CallStaticDoubleMethodA,
    CallStaticVoidMethod,
    CallStaticVoidMethodV,
    CallStaticVoidMethodA,

    GetStaticFieldID,

    GetStaticObjectField,
    GetStaticBooleanField,
    GetStaticByteField,
    GetStaticCharField,
    GetStaticShortField,
    GetStaticIntField,
    GetStaticLongField,
    GetStaticFloatField,
    GetStaticDoubleField,

    SetStaticObjectField,
    SetStaticBooleanField,
    SetStaticByteField,
    SetStaticCharField,
    SetStaticShortField,
    SetStaticIntField,
    SetStaticLongField,
    SetStaticFloatField,
    SetStaticDoubleField,

    NewString,
    GetStringLength,
    GetStringChars,
    ReleaseStringChars,

    NewStringUTF,
    GetStringUTFLength,
    GetStringUTFChars,
    ReleaseStringUTFChars,

    GetArrayLength,

    NewObjectArray,
    GetObjectArrayElement,
    SetObjectArrayElement,
```

Code Example 4-1

```
    NewBooleanArray,
    NewByteArray,
    NewCharArray,
    NewShortArray,
    NewIntArray,
    NewLongArray,
    NewFloatArray,
    NewDoubleArray,

    GetBooleanArrayElements,
    GetByteArrayElements,
    GetCharArrayElements,
    GetShortArrayElements,
    GetIntArrayElements,
    GetLongArrayElements,
    GetFloatArrayElements,
    GetDoubleArrayElements,

    ReleaseBooleanArrayElements,
    ReleaseByteArrayElements,
    ReleaseCharArrayElements,
    ReleaseShortArrayElements,
    ReleaseIntArrayElements,
    ReleaseLongArrayElements,
    ReleaseFloatArrayElements,
    ReleaseDoubleArrayElements,

    GetBooleanArrayRegion,
    GetByteArrayRegion,
    GetCharArrayRegion,
    GetShortArrayRegion,
    GetIntArrayRegion,
    GetLongArrayRegion,
    GetFloatArrayRegion,
    GetDoubleArrayRegion,
    SetBooleanArrayRegion,
    SetByteArrayRegion,
    SetCharArrayRegion,
    SetShortArrayRegion,
    SetIntArrayRegion,
    SetLongArrayRegion,
    SetFloatArrayRegion,
    SetDoubleArrayRegion,
```

Code Example 4-1

```
    RegisterNatives,
    UnregisterNatives,

    MonitorEnter,
    MonitorExit,

    GetJavaVM,
};
```

Version Information

GetVersion

```
jint GetVersion(JNIEnv *env);
```

Returns the version of the native method interface.

PARAMETERS:

`env`: the JNI interface pointer.

RETURNS:

Returns the major version number in the higher 16 bits and the minor version number in the lower 16 bits.

In JDK1.1, `GetVersion()` returns `0x00010001`.

Class Operations

DefineClass

```
jclass DefineClass(JNIEnv *env, jobject loader,
       const jbyte *buf, jsize bufLen);
```

Loads a class from a buffer of raw class data.

PARAMETERS:

`env`: the JNI interface pointer.

`loader`: a class loader assigned to the defined class.

`buf`: buffer containing the `.class` file data.

`bufLen`: buffer length.

RETURNS:

Returns a Java class object or `NULL` if an error occurs.

THROWS:

`ClassFormatError`: if the class data does not specify a valid class.

`ClassCircularityError`: if a class or interface would be its own superclass or superinterface.

`OutOfMemoryError`: if the system runs out of memory.

FindClass

```
jclass FindClass(JNIEnv *env, const char *name);
```

This function loads a locally-defined class. It searches the directories and zip files specified by the `CLASSPATH` environment variable for the class with the specified name.

PARAMETERS:

`env`: the JNI interface pointer.

`name`: a fully-qualified class name (that is, a package name, delimited by "`/`", followed by the class name). If the name begins with "`[`" (the array signature character), it returns an array class.

RETURNS:

Returns a class object from a fully-qualified name, or `NULL` if the class cannot be found.

THROWS:

`ClassFormatError`: if the class data does not specify a valid class.

`ClassCircularityError`: if a class or interface would be its own superclass or superinterface.

`NoClassDefFoundError`: if no definition for a requested class or interface can be found.

`OutOfMemoryError`: if the system runs out of memory.

GetSuperclass

```
jclass GetSuperclass(JNIEnv *env, jclass clazz);
```

If `clazz` represents any class other than the class `Object`, then this function returns the object that represents the superclass of the class specified by `clazz`.

If `clazz` specifies the class `Object`, or `clazz` represents an interface, this function returns `NULL`.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

RETURNS:

Returns the superclass of the class represented by `clazz`, or `NULL`.

IsAssignableFrom

```
jboolean IsAssignableFrom(JNIEnv *env, jclass clazz1,
     jclass clazz2);
```

Determines whether an object of `clazz1` can be safely cast to `clazz2`.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz1`: the first class argument.

`clazz2`: the second class argument.

RETURNS:

Returns `JNI_TRUE` if either of the following is true:

- The first and second class arguments refer to the same Java class.

- The first class is a subclass of the second class.
- The first class has the second class as one of its interfaces.

Exceptions

Throw

```
jint Throw(JNIEnv *env, jthrowable obj);
```

Causes a `java.lang.Throwable` object to be thrown.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a `java.lang.Throwable` object.

RETURNS:

Returns 0 on success; a negative value on failure.

THROWS:

the `java.lang.Throwable` object `obj`.

ThrowNew

```
jint ThrowNew(JNIEnv *env, jclass clazz,
     const char *message);
```

Constructs an exception object from the specified class with the message specified by `message` and causes that exception to be thrown.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a subclass of `java.lang.Throwable`.

`message`: the message used to construct the `java.lang.Throwable` object.

RETURNS:

Returns 0 on success; a negative value on failure.

THROWS:

the newly constructed `java.lang.Throwable` object.

ExceptionOccurred

```
jthrowable ExceptionOccurred(JNIEnv *env);
```

Determines if an exception is being thrown. The exception stays being thrown until either the native code calls `ExceptionClear()`, or the Java code handles the exception.

PARAMETERS:

`env`: the JNI interface pointer.

RETURNS:

Returns the exception object that is currently in the process of being thrown, or `NULL` if no exception is currently being thrown.

ExceptionDescribe

```
void ExceptionDescribe(JNIEnv *env);
```

Prints an exception and a backtrace of the stack to a system error-reporting channel, such as `stderr`. This is a convenience routine provided for debugging.

PARAMETERS:

`env`: the JNI interface pointer.

ExceptionClear

```
void ExceptionClear(JNIEnv *env);
```

Clears any exception that is currently being thrown. If no exception is currently being thrown, this routine has no effect.

PARAMETERS:

`env`: the JNI interface pointer.

FatalError

```
void FatalError(JNIEnv *env, const char *msg);
```

Raises a fatal error and does not expect the VM to recover. This function does not return.

PARAMETERS:

`env`: the JNI interface pointer.

`msg`: an error message.

Global and Local References

NewGlobalRef

```
jobject NewGlobalRef(JNIEnv *env, jobject obj);
```

Creates a new global reference to the object referred to by the `obj` argument. The `obj` argument may be a global or local reference. Global references must be explicitly disposed of by calling `DeleteGlobalRef()`.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a global or local reference.

RETURNS:

Returns a global reference, or `NULL` if the system runs out of memory.

DeleteGlobalRef

```
void DeleteGlobalRef(JNIEnv *env, jobject globalRef);
```

Deletes the global reference pointed to by `globalRef`.

PARAMETERS:

`env`: the JNI interface pointer.

`globalRef`: a global reference.

DeleteLocalRef

```
void DeleteLocalRef(JNIEnv *env, jobject localRef);
```

Deletes the local reference pointed to by `localRef`.

PARAMETERS:

`env`: the JNI interface pointer.

`localRef`: a local reference.

Object Operations

AllocObject

```
jobject AllocObject(JNIEnv *env, jclass clazz);
```

Allocates a new Java object without invoking any of the constructors for the object. Returns a reference to the object.

The `clazz` argument must not refer to an array class.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

RETURNS:

Returns a Java object, or `NULL` if the object cannot be constructed.

THROWS:

`InstantiationException`: if the class is an interface or an abstract class.

`OutOfMemoryError`: if the system runs out of memory.

NewObject
NewObjectA
NewObjectV

```
jobject NewObject(JNIEnv *env, jclass clazz,
      jmethodID methodID, ...);

jobject NewObjectA(JNIEnv *env, jclass clazz,
      jmethodID methodID, jvalue *args);

jobject NewObjectV(JNIEnv *env, jclass clazz,
      jmethodID methodID, va_list args);
```

Constructs a new Java object. The method ID indicates which constructor method to invoke. This ID must be obtained by calling `GetMethodID()` with `<init>` as the method name and `void` (`V`) as the return type.

The `clazz` argument must not refer to an array class.

NewObject

Programmers place all arguments that are to be passed to the constructor immediately following the `methodID` argument. `NewObject()` accepts these arguments and passes them to the Java method that the programmer wishes to invoke.

NewObjectA

Programmers place all arguments that are to be passed to the constructor in an `args` array of `jvalues` that immediately follows the `methodID` argument. `NewObjectA()` accepts the arguments in this array, and, in turn, passes them to the Java method that the programmer wishes to invoke.

NewObjectV

Programmers place all arguments that are to be passed to the constructor in an `args` argument of type `va_list` that immediately follows the `methodID` argument. `NewObjectV()` accepts these arguments, and, in turn, passes them to the Java method that the programmer wishes to invoke.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`methodID`: the method ID of the constructor.

Additional Parameter for NewObject:

arguments to the constructor.

Additional Parameter for NewObjectA:

`args`: an array of arguments to the constructor.

Additional Parameter for NewObjectV:

`args`: a `va_list` of arguments to the constructor.

RETURNS:

Returns a Java object, or `NULL` if the object cannot be constructed.

THROWS:

`InstantiationException`: if the class is an interface or an abstract class.

`OutOfMemoryError`: if the system runs out of memory.

Any exceptions thrown by the constructor.

GetObjectClass

```
jclass GetObjectClass(JNIEnv *env, jobject obj);
```

Returns the class of an object.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a Java object (must not be `NULL`).

RETURNS:

Returns a Java class object.

IsInstanceOf

```
jboolean IsInstanceOf(JNIEnv *env, jobject obj,
      jclass clazz);
```

Tests whether an object is an instance of a class.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a Java object.

`clazz`: a Java class object.

RETURNS:

`Returns JNI_TRUE` if `obj` can be cast to `clazz`; otherwise, returns `JNI_FALSE`. A `NULL` object can be cast to any class.

IsSameObject

```
jboolean IsSameObject(JNIEnv *env, jobject ref1,
      jobject ref2);
```

Tests whether two references refer to the same Java object.

PARAMETERS:

`env`: the JNI interface pointer.

`ref1`: a Java object.

`ref2`: a Java object.

RETURNS:

`Returns JNI_TRUE` if `ref1` and `ref2` refer to the same Java object, or are both `NULL`; otherwise, returns `JNI_FALSE`.

Accessing Fields of Objects

GetFieldID

```
jfieldID GetFieldID(JNIEnv *env, jclass clazz,
      const char *name, const char *sig);
```

Returns the field ID for an instance (nonstatic) field of a class. The field is specified by its name and signature. The *Get<type>Field* and *Set<type>Field* families of accessor functions use field IDs to retrieve object fields.

`GetFieldID()` causes an uninitialized class to be initialized.

`GetFieldID()` cannot be used to obtain the length field of an array. Use `GetArrayLength()` instead.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`name`: the field name in a 0-terminated UTF-8 string.

`sig`: the field signature in a 0-terminated UTF-8 string.

RETURNS:

Returns a field ID, or `NULL` if the operation fails.

THROWS:

`NoSuchFieldError`: if the specified field cannot be found.

`ExceptionInInitializerError`: if the class initializer fails due to an exception.

`OutOfMemoryError`: if the system runs out of memory.

Get<type>Field Routines

```
NativeType Get<type>Field(JNIEnv *env, jobject obj,
      jfieldID fieldID);
```

This family of accessor routines returns the value of an instance (nonstatic) field of an object. The field to access is specified by a field ID obtained by calling `GetFieldID()`.

The following table describes the *Get<type>Field* routine name and result type. You should replace *type* in *Get<type>Field* with the Java type of the field, or use one of the actual routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-1 Get<type>Field Family of Accessor Routines

Get<type>Field Routine Name	Native Type
`GetObjectField()`	`jobject`
`GetBooleanField()`	`jboolean`
`GetByteField()`	`jbyte`
`GetCharField()`	`jchar`
`GetShortField()`	`jshort`
`GetIntField()`	`jint`
`GetLongField()`	`jlong`
`GetFloatField()`	`jfloat`
`GetDoubleField()`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a Java object (must not be `NULL`).

`fieldID`: a valid field ID.

RETURNS:

Returns the content of the field.

Set<type>Field Routines

```
void Set<type>Field(JNIEnv *env, jobject obj, jfieldID fieldID,
     NativeType value);
```

This family of accessor routines sets the value of an instance (nonstatic) field of an object. The field to access is specified by a field ID obtained by calling `GetFieldID()`.

The following table describes the *Set<type>Field* routine name and value type. You should replace *type* in *Set<type>Field* with the Java type of the field, or use one of the actual routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-2 Set<type>Field Family of Accessor Routines

Set<type>Field Routine	**Native Type**
`SetObjectField()`	`jobject`
`SetBooleanField()`	`jboolean`
`SetByteField()`	`jbyte`
`SetCharField()`	`jchar`
`SetShortField()`	`jshort`
`SetIntField()`	`jint`
`SetLongField()`	`jlong`
`SetFloatField()`	`jfloat`
`SetDoubleField()`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a Java object (must not be `NULL`).

`fieldID`: a valid field ID.

`value`: the new value of the field.

Calling Instance Methods

GetMethodID

```
jmethodID GetMethodID(JNIEnv *env, jclass clazz,
      const char *name, const char *sig);
```

Returns the method ID for an instance (nonstatic) method of a class or interface. The method may be defined in one of the `clazz`'s superclasses and inherited by `clazz`. The method is determined by its name and signature.

`GetMethodID()` causes an uninitialized class to be initialized.

To obtain the method ID of a constructor, supply `<init>` as the method name and `void` (`V`) as the return type.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`name`: the method name in a 0-terminated UTF-8 string.

`sig`: the method signature in 0-terminated UTF-8 string.

RETURNS:

Returns a method ID, or `NULL` if the specified method cannot be found.

THROWS:

`NoSuchMethodError`: if the specified method cannot be found.

`ExceptionInInitializerError`: if the class initializer fails due to an exception.

`OutOfMemoryError`: if the system runs out of memory.

Call<type>Method Routines
Call<type>MethodA Routines
Call<type>MethodV Routines

NativeType Call<type>Method `(JNIEnv *env, jobject obj, jmethodID methodID, ...);`

NativeType Call<type>MethodA `(JNIEnv *env, jobject obj, jmethodID methodID, jvalue *args);`

NativeType Call<type>MethodV `(JNIEnv *env, jobject obj, jmethodID methodID, va_list args);`

Methods from these three families of operations are used to call a Java instance method from a native method.They only differ in their mechanism for passing parameters to the methods that they call.

These families of operations invoke an instance (nonstatic) method on a Java object, according to the specified method ID. The `methodID` argument must be obtained by calling `GetMethodID()`.

When these functions are used to call private methods and constructors, the method ID must be derived from the real class of `obj`, not from one of its superclasses.

Call<type>Method Routines

Programmers place all arguments that are to be passed to the method immediately following the `methodID` argument. The *Call<type>Method* routine accepts these arguments and passes them to the Java method that the programmer wishes to invoke.

Call<type>MethodA Routines

Programmers place all arguments to the method in an `args` array of `jvalues` that immediately follows the `methodID` argument. The *Call<type>MethodA* routine accepts the arguments in this array, and, in turn, passes them to the Java method that the programmer wishes to invoke.

Call<type>MethodV Routines

Programmers place all arguments to the method in an `args` argument of type `va_list` that immediately follows the `methodID` argument. The

Call<type>MethodV routine accepts the arguments, and, in turn, passes them to the Java method that the programmer wishes to invoke.

The following table describes each of the method calling routines according to their result type. You should replace *type* in *Call<type>Method* with the Java type of the method you are calling (or use one of the actual method calling routine names from the table) and replace *NativeType* with the corresponding native type for that routine.

Table 4-3 Instance Method Calling Routines

Call<type>Method Routine Name	**Native Type**

`CallVoidMethod()`
`CallVoidMethodA()`
`CallVoidMethodV()`
	`void`

`CallObjectMethod()`
`CallObjectMethodA()`
`CallObjectMethodV()`
	`jobject`

`CallBooleanMethod()`
`CallBooleanMethodA()`
`CallBooleanMethodV()`
	`jboolean`

`CallByteMethod()`
`CallByteMethodA()`
`CallByteMethodV()`
	`jbyte`

`CallCharMethod()`
`CallCharMethodA()`
`CallCharMethodV()`
	`jchar`

`CallShortMethod()`
`CallShortMethodA()`
`CallShortMethodV()`
	`jshort`

`CallIntMethod()`
`CallIntMethodA()`
`CallIntMethodV()`
	`jint`

Table 4-3 Instance Method Calling Routines

Call<type>Method Routine Name	Native Type

`CallLongMethod()`
`CallLongMethodA()`
`CallLongMethodV()`
	`jlong`

`CallFloatMethod()`
`CallFloatMethodA()`
`CallFloatMethodV()`
	`jfloat`

`CallDoubleMethod()`
`CallDoubleMethodA()`
`CallDoubleMethodV()`
	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a Java object.

`methodID`: a method ID.

Additional Parameter for Call<type>Method Routines:

arguments to the Java method.

Additional Parameter for Call<type>MethodA Routines:

`args`: an array of arguments.

Additional Parameter for Call<type>MethodV Routines:

`args`: a `va_list` of arguments.

RETURNS:

Returns the result of calling the Java method.

THROWS:

Exceptions raised during the execution of the Java method.

CallNonvirtual<type>Method Routines
CallNonvirtual<type>MethodA Routines
CallNonvirtual<type>MethodV Routines

```
NativeType CallNonvirtual<type>Method(JNIEnv *env, jobject obj,
      jclass clazz, jmethodID methodID, ...);

NativeType CallNonvirtual<type>MethodA(JNIEnv *env, jobject obj,
      jclass clazz, jmethodID methodID, jvalue *args);

NativeType CallNonvirtual<type>MethodV(JNIEnv *env, jobject obj,
      jclass clazz, jmethodID methodID, va_list args);
```

These families of operations invoke an instance (nonstatic) method on a Java object, according to the specified class and method ID. The `methodID` argument must be obtained by calling `GetMethodID()` on the class `clazz`.

The *CallNonvirtual<type>Method* families of routines and the *Call<type>Method* families of routines are different. *Call<type>Method* routines invoke the method based on the class of the object, while *CallNonvirtual<type>Method* routines invoke the method based on the class, designated by the `clazz` parameter, from which the method ID is obtained. The method ID must be obtained from the real class of the object or from one of its superclasses.

CallNonvirtual<type>Method Routines

Programmers place all arguments that are to be passed to the method immediately following the `methodID` argument. The *CallNonvirtual<type>Method* routine accepts these arguments and passes them to the Java method that the programmer wishes to invoke.

CallNonvirtual<type>MethodA Routines

Programmers place all arguments to the method in an `args` array of `jvalues` that immediately follows the `methodID` argument. The *CallNonvirtual<type>MethodA* routine accepts the arguments in this array, and, in turn, passes them to the Java method that the programmer wishes to invoke.

CallNonvirtual<type>MethodV Routines

Programmers place all arguments to the method in an `args` argument of type `va_list` that immediately follows the `methodID` argument. The

CallNonvirtualMethodV routine accepts the arguments, and, in turn, passes them to the Java method that the programmer wishes to invoke.

The following table describes each of the method calling routines according to their result type. You should replace *type* in *CallNonvirtual<type>Method* with the Java type of the method, or use one of the actual method calling routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-4 CallNonvirtual<type>Method Routines

CallNonvirtual<type>Method Routine Name	**Native Type**

`CallNonvirtualVoidMethod()`
`CallNonvirtualVoidMethodA()`
`CallNonvirtualVoidMethodV()`
	`void`

`CallNonvirtualObjectMethod()`
`CallNonvirtualObjectMethodA()`
`CallNonvirtualObjectMethodV()`
	`jobject`

`CallNonvirtualBooleanMethod()`
`CallNonvirtualBooleanMethodA()`
`CallNonvirtualBooleanMethodV()`
	`jboolean`

`CallNonvirtualByteMethod()`
`CallNonvirtualByteMethodA()`
`CallNonvirtualByteMethodV()`
	`jbyte`

`CallNonvirtualCharMethod()`
`CallNonvirtualCharMethodA()`
`CallNonvirtualCharMethodV()`
	`jchar`

`CallNonvirtualShortMethod()`
`CallNonvirtualShortMethodA()`
`CallNonvirtualShortMethodV()`
	`jshort`

`CallNonvirtualIntMethod()`
`CallNonvirtualIntMethodA()`
`CallNonvirtualIntMethodV()`
	`jint`

Table 4-4 CallNonvirtual<type>Method Routines

CallNonvirtual<type>Method Routine Name	Native Type

`CallNonvirtualLongMethod()`
`CallNonvirtualLongMethodA()`
`CallNonvirtualLongMethodV()`
	`jlong`

`CallNonvirtualFloatMethod()`
`CallNonvirtualFloatMethodA()`
`CallNonvirtualFloatMethodV()`
	`jfloat`

`CallNonvirtualDoubleMethod()`
`CallNonvirtualDoubleMethodA()`
`CallNonvirtualDoubleMethodV()`
	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class.

`obj`: a Java object.

`methodID`: a method ID.

Additional Parameter for CallNonvirtual<type>Method Routines:

arguments to the Java method.

Additional Parameter for CallNonvirtual<type>MethodA Routines:

`args`: an array of arguments.

Additional Parameter for CallNonvirtual<type>MethodV Routines:

`args`: a `va_list` of arguments.

RETURNS:

Returns the result of calling the Java method.

THROWS:

Exceptions raised during the execution of the Java method.

Accessing Static Fields

GetStaticFieldID

```
jfieldID GetStaticFieldID(JNIEnv *env, jclass clazz,
      const char *name, const char *sig);
```

Returns the field ID for a static field of a class. The field is specified by its name and signature. The *GetStatic<type>Field* and *SetStatic<type>Field* families of accessor functions use field IDs to retrieve static fields.

`GetStaticFieldID()` causes an uninitialized class to be initialized.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`name`: the static field name in a 0-terminated UTF-8 string.

`sig`: the field signature in a 0-terminated UTF-8 string.

RETURNS:

Returns a field ID, or `NULL` if the specified static field cannot be found.

THROWS:

`NoSuchFieldError`: if the specified static field cannot be found.

`ExceptionInInitializerError`: if the class initializer fails due to an exception.

`OutOfMemoryError`: if the system runs out of memory.

GetStatic<type>Field Routines

NativeType GetStatic<type>Field`(JNIEnv *env, jclass clazz,`
`      jfieldID fieldID);`

This family of accessor routines returns the value of a static field of an object. The field to access is specified by a field ID, which is obtained by calling `GetStaticFieldID()`.

The following table describes the family of get routine names and result types. You should replace *type* in *GetStatic<type>Field* with the Java type of the field, or one of the actual static field accessor routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-5 GetStatic<type>Field Family of Accessor Routines

GetStatic<type>Field Routine Name	Native Type
`GetStaticObjectField()`	`jobject`
`GetStaticBooleanField()`	`jboolean`
`GetStaticByteField()`	`jbyte`
`GetStaticCharField()`	`jchar`
`GetStaticShortField()`	`jshort`
`GetStaticIntField()`	`jint`
`GetStaticLongField()`	`jlong`
`GetStaticFloatField()`	`jfloat`
`GetStaticDoubleField()`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`fieldID`: a static field ID.

RETURNS:

Returns the content of the static field.

SetStatic<type>Field Routines

```
void SetStatic<type>Field(JNIEnv *env, jclass clazz,
       jfieldID fieldID, NativeType value);
```

This family of accessor routines sets the value of a static field of an object. The field to access is specified by a field ID, which is obtained by calling `GetStaticFieldID()`.

The following table describes the set routine name and value types. You should replace *type* in *SetStatic<type>Field* with the Java type of the field, or one of the actual set static field routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-6 SetStatic<type>Field Family of Accessor Routines

SetStatic<type>Field Routine Name	NativeType
`SetStaticObjectField()`	`jobject`
`SetStaticBooleanField()`	`jboolean`
`SetStaticByteField()`	`jbyte`
`SetStaticCharField()`	`jchar`
`SetStaticShortField()`	`jshort`
`SetStaticIntField()`	`jint`
`SetStaticLongField()`	`jlong`
`SetStaticFloatField()`	`jfloat`
`SetStaticDoubleField()`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`fieldID`: a static field ID.

`value`: the new value of the field.

Calling Static Methods

GetStaticMethodID

```
jmethodID GetStaticMethodID(JNIEnv *env, jclass clazz,
      const char *name, const char *sig);
```

Returns the method ID for a static method of a class. The method is specified by its name and signature.

`GetStaticMethodID()` causes an uninitialized class to be initialized.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`name`: the static method name in a 0-terminated UTF-8 string.

`sig`: the method signature in a 0-terminated UTF-8 string.

RETURNS:

Returns a method ID, or `NULL` if the operation fails.

THROWS:

`NoSuchMethodError`: if the specified static method cannot be found.

`ExceptionInInitializerError`: if the class initializer fails due to an exception.

`OutOfMemoryError`: if the system runs out of memory.

CallStatic<type>Method Routines
 CallStatic<type>MethodA Routines
 CallStatic<type>MethodV Routines

NativeType CallStatic<type>Method `(JNIEnv *env, jclass clazz, jmethodID methodID, ...);`

NativeType CallStatic<type>MethodA `(JNIEnv *env, jclass clazz, jmethodID methodID, jvalue *args);`

NativeType CallStatic<type>MethodV `(JNIEnv *env, jclass clazz, jmethodID methodID, va_list args);`

This family of operations invokes a static method on a Java object, according to the specified method ID. The `methodID` argument must be obtained by calling `GetStaticMethodID()`.

The method ID must be derived from `clazz`, not from one of its superclasses.

CallStatic<type>Method Routines

Programmers should place all arguments that are to be passed to the method immediately following the `methodID` argument. The *CallStatic<type>Method* routine accepts these arguments and passes them to the Java method that the programmer wishes to invoke.

CallStatic<type>MethodA Routines

Programmers should place all arguments to the method in an `args` array of `jvalues` that immediately follows the `methodID` argument. The *CallStaticMethodA* routine accepts the arguments in this array, and, in turn, passes them to the Java method that the programmer wishes to invoke.

CallStatic<type>MethodV Routines

Programmers should place all arguments to the method in an `args` argument of type `va_list` that immediately follows the `methodID` argument. The *CallStaticMethodV* routine accepts the arguments, and, in turn, passes them to the Java method that the programmer wishes to invoke.

The following table describes each of the method calling routines according to their result types. You should replace *type* in *CallStatic<type>Method* with the Java type of the method, or one of the actual method calling routine names from the table, and replace *NativeType* with the corresponding native type for that routine.

Table 4-7 CallStatic<type>Method Calling Routines

CallStatic<type>Method Routine Name	**Native Type**

`CallStaticVoidMethod()`
`CallStaticVoidMethodA()`
`CallStaticVoidMethodV()`
	`void`

`CallStaticObjectMethod()`
`CallStaticObjectMethodA()`
`CallStaticObjectMethodV()`
	`jobject`

`CallStaticBooleanMethod()`
`CallStaticBooleanMethodA()`
`CallStaticBooleanMethodV()`
	`jboolean`

`CallStaticByteMethod()`
`CallStaticByteMethodA()`
`CallStaticByteMethodV()`
	`jbyte`

Table 4-7 CallStatic<type>Method Calling Routines

CallStatic<type>Method Routine Name	Native Type

`CallStaticCharMethod()`
`CallStaticCharMethodA()`
`CallStaticCharMethodV()`
	`jchar`

`CallStaticShortMethod()`
`CallStaticShortMethodA()`
`CallStaticShortMethodV()`
	`jshort`

`CallStaticIntMethod()`
`CallStaticIntMethodA()`
`CallStaticIntMethodV()`
	`jint`

`CallStaticLongMethod()`
`CallStaticLongMethodA()`
`CallStaticLongMethodV()`
	`jlong`

`CallStaticFloatMethod()`
`CallStaticFloatMethodA()`
`CallStaticFloatMethodV()`
	`jfloat`

`CallStaticDoubleMethod()`
`CallStaticDoubleMethodA()`
`CallStaticDoubleMethodV()`
	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`methodID`: a static method ID.

Additional Parameter for CallStatic<type>Method Routines:

arguments to the static method.

Additional Parameter for CallStatic<type>MethodA Routines:

`args`: an array of arguments.

Additional Parameter for CallStatic<type>MethodV Routines:

`args`: a `va_list` of arguments.

RETURNS:

Returns the result of calling the static Java method.

THROWS:

Exceptions raised during the execution of the Java method.

String Operations

NewString

```
jstring NewString(JNIEnv *env, const jchar *unicodeChars,
      jsize len);
```

Constructs a new `java.lang.String` object from an array of Unicode characters.

PARAMETERS:

`env`: the JNI interface pointer.

`unicodeChars`: pointer to a Unicode string.

`len`: length of the Unicode string.

RETURNS:

Returns a Java string object, or `NULL` if the string cannot be constructed.

THROWS:

`OutOfMemoryError`: if the system runs out of memory.

GetStringLength

```
jsize GetStringLength(JNIEnv *env, jstring string);
```

Returns the length (the count of Unicode characters) of a Java string.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

RETURNS:

Returns the length of the Java string.

GetStringChars

```
const jchar * GetStringChars(JNIEnv *env, jstring string,
      jboolean *isCopy);
```

Returns a pointer to the array of Unicode characters of the string. This pointer is valid until `ReleaseStringchars()` is called.

If `isCopy` is not `NULL`, then `*isCopy` is set to `JNI_TRUE` if a copy is made; or it is set to `JNI_FALSE` if no copy is made.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

`isCopy`: a pointer to a boolean.

RETURNS:

Returns a pointer to a Unicode string, or `NULL` if the operation fails.

ReleaseStringChars

```
void ReleaseStringChars(JNIEnv *env, jstring string,
      const jchar *chars);
```

Informs the VM that the native code no longer needs access to `chars`. The `chars` argument is a pointer obtained from `string` using `GetStringChars()`.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

`chars`: a pointer to a Unicode string.

NewStringUTF

```
jstring NewStringUTF(JNIEnv *env, const char *bytes);
```

Constructs a new `java.lang.String` object from an array of UTF-8 characters.

PARAMETERS:

`env`: the JNI interface pointer, or `NULL` if the string cannot be constructed.

`bytes`: the pointer to a UTF-8 string.

RETURNS:

Returns a Java string object, or `NULL` if the string cannot be constructed.

THROWS:

`OutOfMemoryError`: if the system runs out of memory.

GetStringUTFLength

```
jsize GetStringUTFLength(JNIEnv *env, jstring string);
```

Returns the UTF-8 length in bytes of a string.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

RETURNS:

Returns the UTF-8 length of the string.

GetStringUTFChars

```
const jbyte* GetStringUTFChars(JNIEnv *env, jstring string,
        jboolean *isCopy);
```

Returns a pointer to an array of UTF-8 characters of the string. This array is valid until it is released by `ReleaseStringUTFChars()`.

If isCopy is not NULL, then *isCopy is set to JNI_TRUE if a copy is made; or it is set to JNI_FALSE if no copy is made.

PARAMETERS:

env: the JNI interface pointer.

string: a Java string object.

isCopy: a pointer to a boolean.

RETURNS:

Returns a pointer to a UTF-8 string, or NULL if the operation fails.

ReleaseStringUTFChars

```
void ReleaseStringUTFChars(JNIEnv *env, jstring string,
     const char *utf);
```

Informs the VM that the native code no longer needs access to utf. The utf argument is a pointer derived from string using GetStringUTFChars().

PARAMETERS:

env: the JNI interface pointer.

string: a Java string object.

utf: a pointer to a UTF-8 string.

Array Operations

GetArrayLength

```
jsize GetArrayLength(JNIEnv *env, jarray array);
```

Returns the number of elements in the array.

PARAMETERS:

env: the JNI interface pointer.

array: a Java array object.

RETURNS:

Returns the length of the array.

NewObjectArray

```
jarray NewObjectArray(JNIEnv *env, jsize length,
      jclass elementClass, jobject initialElement);
```

Constructs a new array holding objects in class `elementClass`. All elements are initially set to `initialElement`.

PARAMETERS:

`env`: the JNI interface pointer.

`length`: array size.

`elementClass`: array element class.

`initialElement`: initialization value.

RETURNS:

Returns a Java array object, or `NULL` if the array cannot be constructed.

THROWS:

`OutOfMemoryError`: if the system runs out of memory.

GetObjectArrayElement

```
jobject GetObjectArrayElement(JNIEnv *env,
      jobjectArray array, jsize index);
```

Returns an element of an `Object` array.

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array.

`index`: array index.

RETURNS:

Returns a Java object.

THROWS:

`ArrayIndexOutOfBoundsException`: if `index` does not specify a valid index in the array.

SetObjectArrayElement

```
void SetObjectArrayElement(JNIEnv *env, jobjectArray array,
     jsize index, jobject value);
```

Sets an element of an `Object` array.

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array.

`index`: array index.

`value`: the new value.

THROWS:

`ArrayIndexOutOfBoundsException`: if `index` does not specify a valid index in the array.

`ArrayStoreException`: if the class of `value` is not a subclass of the element class of the array.

New<PrimitiveType>Array Routines

ArrayType *New<PrimitiveType>Array*`(JNIEnv *env, jsize length);`

A family of operations used to construct a new primitive array object. Table 4-8 describes the specific primitive array constructors. You should replace *New<PrimitiveType>Array* with one of the actual primitive array constructor routine names from the following table, and replace *ArrayType* with the corresponding array type for that routine.

Table 4-8 New<PrimitiveType>Array Family of Array Constructors

New<PrimitiveType>Array Routines	Array Type
`NewBooleanArray()`	`jbooleanArray`
`NewByteArray()`	`jbyteArray`
`NewCharArray()`	`jcharArray`
`NewShortArray()`	`jshortArray`
`NewIntArray()`	`jintArray`
`NewLongArray()`	`jlongArray`
`NewFloatArray()`	`jfloatArray`
`NewDoubleArray()`	`jdoubleArray`

PARAMETERS:

`env`: the JNI interface pointer.

`length`: the array length.

RETURNS:

Returns a Java array, or `NULL` if the array cannot be constructed.

Get<PrimitiveType>ArrayElements Routines

```
NativeType *Get<PrimitiveType>ArrayElements(JNIEnv *env,
       ArrayType array, jboolean *isCopy);
```

A family of functions that returns the body of the primitive array. The result is valid until the corresponding *Release<PrimitiveType>ArrayElements()* function is called. *Since the returned array may be a copy of the Java array, changes made to the returned array will not necessarily be reflected in the original* `array` *until Release<PrimitiveType>ArrayElements() is called.*

If `isCopy` is not `NULL`, then `*isCopy` is set to `JNI_TRUE` if a copy is made; or it is set to `JNI_FALSE` if no copy is made.

The following table describes the specific primitive array element accessors. You should make the following substitutions:

- Replace *Get<PrimitiveType>ArrayElements* with one of the actual primitive element accessor routine names from the table.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

Regardless of how boolean arrays are represented in the Java VM, `GetBooleanArrayElements()` always returns a pointer to `jbooleans`, with each byte denoting an element (the unpacked representation). All arrays of other types are guaranteed to be contiguous in memory.

Table 4-9 Get<PrimitiveType>ArrayElements Family of Accessor Routines

Get<PrimitiveType>ArrayElements Routines	**Array Type**	**Native Type**
`GetBooleanArrayElements()`	`jbooleanArray`	`jboolean`
`GetByteArrayElements()`	`jbyteArray`	`jbyte`
`GetCharArrayElements()`	`jcharArray`	`jchar`
`GetShortArrayElements()`	`jshortArray`	`jshort`
`GetIntArrayElements()`	`jintArray`	`jint`
`GetLongArrayElements()`	`jlongArray`	`jlong`
`GetFloatArrayElements()`	`jfloatArray`	`jfloat`
`GetDoubleArrayElements()`	`jdoubleArray`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java string object.

`isCopy`: a pointer to a boolean.

RETURNS:

Returns a pointer to the array elements, or `NULL` if the operation fails.

Release<PrimitiveType>ArrayElements Routines

```
void Release<PrimitiveType>ArrayElements(JNIEnv *env,
      ArrayType array, NativeType *elems, jint mode);
```

A family of functions that informs the VM that the native code no longer needs access to `elems`. The `elems` argument is a pointer derived from `array` using the corresponding *Get<PrimitiveType>ArrayElements()* function. If necessary, this function copies back all changes made to `elems` to the original `array`.

The `mode` argument provides information on how the array buffer should be released. `mode` has no effect if `elems` is not a copy of the elements in `array`. Otherwise, `mode` has the following impact, as shown in the following table:

Table 4-10 Primitive Array Release Modes

mode	**actions**
`0`	copy back the content and free the `elems` buffer
`JNI_COMMIT`	copy back the content but do not free the `elems` buffer
`JNI_ABORT`	free the buffer without copying back the possible changes

In most cases, programmers pass "0" to the `mode` argument to ensure consistent behavior for both pinned and copied arrays. The other options give the programmer more control over memory management and should be used with extreme care.

The next table describes the specific routines that comprise the family of primitive array disposers. You should make the following substitutions:

- Replace *Release<PrimitiveType>ArrayElements* with one of the actual primitive array disposer routine names from Table 4-11.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

Table 4-11 Release<PrimitiveType>ArrayElements Family of Array Routines

Release<PrimitiveType>ArrayElements Routines	Array Type	Native Type
`ReleaseBooleanArrayElements()`	`jbooleanArray`	`jboolean`
`ReleaseByteArrayElements()`	`jbyteArray`	`jbyte`
`ReleaseCharArrayElements()`	`jcharArray`	`jchar`
`ReleaseShortArrayElements()`	`jshortArray`	`jshort`
`ReleaseIntArrayElements()`	`jintArray`	`jint`
`ReleaseLongArrayElements()`	`jlongArray`	`jlong`
`ReleaseFloatArrayElements()`	`jfloatArray`	`jfloat`
`ReleaseDoubleArrayElements()`	`jdoubleArray`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array object.

`elems`: a pointer to array elements.

`mode`: the release mode.

Get<PrimitiveType>ArrayRegion Routines

void Get<PrimitiveType>ArrayRegion`(JNIEnv *env,` *ArrayType* `array,`
`jsize start, jsize len,` *NativeType* `*buf);`

A family of functions that copies a region of a primitive array into a buffer.

The following table describes the specific primitive array element accessors. You should do the following substitutions:

- Replace *Get<PrimitiveType>ArrayRegion* with one of the actual primitive element accessor routine names from Table 4-12.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

Table 4-12 Get<PrimitiveType>ArrayRegion Family of Array Accessor Routines

Get<PrimitiveType>ArrayRegion Routine	**Array Type**	**Native Type**
`GetBooleanArrayRegion()`	`jbooleanArray`	`jboolean`
`GetByteArrayRegion()`	`jbyteArray`	`jbyte`
`GetCharArrayRegion()`	`jcharArray`	`jchar`
`GetShortArrayRegion()`	`jshortArray`	`jhort`
`GetIntArrayRegion()`	`jintArray`	`jint`
`GetLongArrayRegion()`	`jlongArray`	`jlong`
`GetFloatArrayRegion()`	`jfloatArray`	`jloat`
`GetDoubleArrayRegion()`	`jdoubleArray`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array.

`start`: the starting index.

`len`: the number of elements to be copied.

`buf`: the destination buffer.

THROWS:

`ArrayIndexOutOfBoundsException`: if one of the indexes in the region is not valid.

Set<PrimitiveType>ArrayRegion Routines

```
void Set<PrimitiveType>ArrayRegion(JNIEnv *env, ArrayType array,
        jsize start, jsize len, NativeType *buf);
```

A family of functions that copies back a region of a primitive array from a buffer.

The following table describes the specific primitive array element accessors. You should make the following replacements:

- Replace *Set<PrimitiveType>ArrayRegion* with one of the actual primitive element accessor routine names from the table.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

Table 4-13 Set<PrimitiveType>ArrayRegion Family of Array Accessor Routines

Set<PrimitiveType>ArrayRegion Routine	**Array Type**	**Native Type**
`SetBooleanArrayRegion()`	`jbooleanArray`	`jboolean`
`SetByteArrayRegion()`	`jbyteArray`	`jbyte`
`SetCharArrayRegion()`	`jcharArray`	`jchar`
`SetShortArrayRegion()`	`jshortArray`	`jshort`
`SetIntArrayRegion()`	`jintArray`	`jint`
`SetLongArrayRegion()`	`jlongArray`	`jlong`
`SetFloatArrayRegion()`	`jfloatArray`	`jfloat`
`SetDoubleArrayRegion()`	`jdoubleArray`	`jdouble`

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array.

`start`: the starting index.

`len`: the number of elements to be copied.

`buf`: the destination buffer.

THROWS:

`ArrayIndexOutOfBoundsException`: if one of the indexes in the region is not valid.

Registering Native Methods

RegisterNatives

```
jint RegisterNatives(JNIEnv *env, jclass clazz,
      const JNINativeMethod *methods, jint nMethods);
```

Registers native methods with the class specified by the `clazz` argument. The `methods` parameter specifies an array of `JNINativeMethod` structures that contain the names, signatures, and function pointers of the native methods. The `nMethods` parameter specifies the number of native methods in the array. The `JNINativeMethod` structure is defined as follows:

```
typedef struct {
    char *name;
    char *signature;
    void *fnPtr;
} JNINativeMethod;
```

The function pointers nominally must have the following signature:

```
ReturnType (*fnPtr)(JNIEnv *env, jobject objectOrClass, ...);
```

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

`methods`: the native methods in the class.

`nMethods`: the number of native methods in the class.

RETURNS:

Returns “0” on success; returns a negative value on failure.

THROWS:

`NoSuchMethodError`: if a specified method cannot be found or if the method is not native.

UnregisterNatives

```
jint UnregisterNatives(JNIEnv *env, jclass clazz);
```

Unregisters native methods of a class. The class goes back to the state before it was linked or registered with its native method functions.

This function should not be used in normal native code. Instead, it provides special programs a way to reload and relink native libraries.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

RETURNS:

Returns "0" on success; returns a negative value on failure.

Monitor Operations

MonitorEnter

```
jint MonitorEnter(JNIEnv *env, jobject obj);
```

Enters the monitor associated with the underlying Java object referred to by `obj`.

Each Java object has a monitor associated with it. If the current thread already owns the monitor associated with `obj`, it increments a counter in the monitor indicating the number of times this thread has entered the monitor. If the monitor associated with `obj` is not owned by any thread, the current thread becomes the owner of the monitor, setting the entry count of this monitor to 1. If another thread already owns the monitor associated with `obj`, the current thread waits until the monitor is released, then tries again to gain ownership.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a normal Java object or class object.

RETURNS:

Returns "0" on success; returns a negative value on failure.

MonitorExit

```
jint MonitorExit(JNIEnv *env, jobject obj);
```

The current thread must be the owner of the monitor associated with the underlying Java object referred to by `obj`. The thread decrements the counter indicating the number of times it has entered this monitor. If the value of the counter becomes zero, the current thread releases the monitor.

PARAMETERS:

`env`: the JNI interface pointer.

`obj`: a normal Java object or class object.

RETURNS:

Returns "0" on success; returns a negative value on failure.

Java VM Interface

GetJavaVM

```
jint GetJavaVM(JNIEnv *env, JavaVM **vm);
```

Returns the Java VM interface (used in the Invocation API) associated with the current thread. The result is placed at the location pointed to by the second argument, `vm`.

PARAMETERS:

`env`: the JNI interface pointer.

`vm`: a pointer to where the result should be placed.

RETURNS:

Returns "0" on success; returns a negative value on failure.

The Invocation API 5

The Invocation API allows software vendors to load the Java VM into an arbitrary native application. Vendors can deliver Java-enabled applications without having to link with the Java VM source code.

This chapter begins with an overview of the Invocation API. This is followed by reference pages for all Invocation API functions.

To enhance the embeddability of the Java VM, the Invocation API is extended in JDK 1.1.2 in a few minor ways.

Overview

The following code example illustrates how to use functions in the Invocation API. In this example, the C++ code creates a Java VM and invokes a static method, called `Main.test`. For clarity, we omit error checking.

```
#include <jni.h>       /* where everything is defined */

...

JavaVM *jvm;       /* denotes a Java VM */
JNIEnv *env;       /* pointer to native method interface */

JDK1_1InitArgs vm_args; /* JDK 1.1 VM initialization arguments */

vm_args.version = 0x00010001; /* New in 1.1.2: VM version */
/* Get the default initialization arguments and set the class
 * path */
JNI_GetDefaultJavaVMInitArgs(&vm_args);
vm_args.classpath = ...;
```

```
/* load and initialize a Java VM, return a JNI interface
 * pointer in env */
JNI_CreateJavaVM(&jvm, &env, &vm_args);

/* invoke the Main.test method using the JNI */
jclass cls = env->FindClass("Main");
jmethodID mid = env->GetStaticMethodID(cls, "test", "(I)V");
env->CallStaticVoidMethod(cls, mid, 100);

/* We are done. */
jvm->DestroyJavaVM();
```

This example uses three functions in the API. The Invocation API allows a native application to use the JNI interface pointer to access VM features. The design is similar to Netscape's JRI Embedding Interface.

Creating the VM

The `JNI_CreateJavaVM()` function loads and initializes a Java VM and returns a pointer to the JNI interface pointer. The thread that called `JNI_CreateJavaVM()` is considered to be the *main thread*.

Attaching to the VM

The JNI interface pointer (`JNIEnv`) is valid only in the current thread. Should another thread need to access the Java VM, it must first call `AttachCurrentThread()` to attach itself to the VM and obtain a JNI interface pointer. Once attached to the VM, a native thread works just like an ordinary Java thread running inside a native method. The native thread remains attached to the VM until it calls `DetachCurrentThread()` to detach itself.

Unloading the VM

The main thread cannot detach itself from the VM. Instead, it must call `DestroyJavaVM()` to unload the entire VM.

The VM waits until the main thread is the only *user thread* before it actually unloads. User threads include both Java threads and attached native threads. This restriction exists because a Java thread or attached native thread may be holding system resources, such as locks, windows, and so on. The `VM` cannot automatically free these resources. By restricting the main thread to be the only running thread when the VM is unloaded, the burden of releasing system resources held by arbitrary threads is on the programmer.

Initialization Structures

Different Java VM implementation will likely require different initialization arguments. It is difficult to come up with a standard initialization structure suitable for all present and future Java VMs. As a compromise, we reserve the first field (`version`) to identify the content of the initialization structure. Native applications embedding JDK 1.1.2 must set the version field to `0x00010001`. VM implementations are encouraged to use the same initialization structure as JDK, although other implementations may choose to ignore some of the initialization arguments supported by JDK.

Version numbers `0x80000000` to `0xFFFFFFFF` are reserved, and should not be recognized by any VM implementation.

The following code shows the structure used to initialize the Java VM in JDK 1.1.2.

```
typedef struct JDK1_1InitArgs {
   /* The first two fields were reserved in JDK 1.1, and
      formally introduced in JDK 1.1.2. */
   /* Java VM version */
    jint version;

   /* System properties. */
    char **properties;

   /* whether to check the Java source files are newer than
    * compiled class files. */
    jint checkSource;

   /* maximum native stack size of Java-created threads. */
    jint nativeStackSize;

   /* maximum Java stack size. */
    jint javaStackSize;

   /* initial heap size. */
    jint minHeapSize;

   /* maximum heap size. */
    jint maxHeapSize;
```

```
    /* controls whether Java byte code should be verified:
     * 0 -- none, 1 -- remotely loaded code, 2 -- all code. */
     jint verifyMode;

    /* the local directory path for class loading. */
     const char *classpath;

    /* a hook for a function that redirects all VM messages. */
     jint (*vfprintf)(FILE *fp, const char *format,
                      va_list args);

    /* a VM exit hook. */
     void (*exit)(jint code);

    /* a VM abort hook. */
     void (*abort)();

    /* whether to enable class GC. */
     jint enableClassGC;

    /* whether GC messages will appear. */
     jint enableVerboseGC;

    /* whether asynchronous GC is allowed. */
     jint disableAsyncGC;

    /* Three reserved fields. */
     jint reserved0;
     jint reserved1;
     jint reserved2;
} JDK1_1InitArgs;
```

In JDK 1.1.2, the initialization structure provides hooks so that a native application can redirect VM messages and obtain control when the VM terminates.

The structure below is passed as an argument when a native thread attaches to a Java VM in JDK 1.1.2. In actuality, no arguments are required for a native thread to attach to the JDK 1.1.2. The `JDK1_1AttachArgs` structure consists only of a padding slot for those C compilers that do not permit empty structures.

```
typedef struct JDK1_1AttachArgs {
    /*
     * JDK 1.1 does not need any arguments to attach a
```

```
    * native thread. The padding is here to satisfy the C
    * compiler which does not permit empty structures.
    */
    void *__padding;
} JDK1_1AttachArgs;
```

Invocation API Functions

The JavaVM type is a pointer to the Invocation API function table. The following code example shows this function table.

```
typedef const struct JNIInvokeInterface *JavaVM;

const struct JNIInvokeInterface ... = {
    NULL,
    NULL,
    NULL,

    DestroyJavaVM,
    AttachCurrentThread,
    DetachCurrentThread,
};
```

Note that three Invocation API functions, `JNI_GetDefaultJavaVMInitArgs()`, `JNI_GetCreatedJavaVMs()`, and `JNI_CreateJavaVM()`, are not part of the JavaVM function table. These functions can be used without a preexisting `JavaVM` structure.

JNI_GetDefaultJavaVMInitArgs

```
jint JNI_GetDefaultJavaVMInitArgs(void *vm_args);
```

Returns a default configuration for the Java VM. Before calling this function, native code must[*] set the `vm_args->version` field to the JNI version it expects the VM to support. In JDK 1.1.2, `vm_args->version` must be set to `0x00010001`. After this function returns, `vm_args->version` will be set to the actual JNI version the VM supports.

* JDK 1.1 did not require the native code to set the version field. For backward compatibility, JDK 1.1.2 assumes that the requested version is 0x00010001 if the version field is not set. Future versions of JDK will require the version field to be set to an appropriate value.

PARAMETERS:

`vm_args`: a pointer to a VM-specific initialization structure in to which the default arguments are filled.

RETURNS:

Returns “0” if the requested version is supported; returns a negative number if the requested version is not supported.

JNI_GetCreatedJavaVMs

```
jint JNI_GetCreatedJavaVMs(JavaVM **vmBuf, jsize bufLen,
      jsize *nVMs);
```

Returns all Java VMs that have been created. Pointers to VMs are written in the buffer `vmBuf` in the order they are created. At most `bufLen` number of entries will be written. The total number of created VMs is returned in `*nVMs`.

JDK 1.1.2 does not support creating more than one VM in a single process.

PARAMETERS:

`vmBuf`: pointer to the buffer where the VM structures will be placed.

`bufLen`: the length of the buffer.

`nVMs`: a pointer to an integer.

RETURNS:

Returns “0” on success; returns a negative number on failure.

JNI_CreateJavaVM

```
jint JNI_CreateJavaVM(JavaVM **p_vm, JNIEnv **p_env,
      void *vm_args);
```

Loads and initializes a Java VM. The current thread becomes the main thread. Sets the `env` argument to the JNI interface pointer of the main thread.

JDK 1.1 does not support creating more than one VM in a single process. The version field in `vm_args` must* be set to `0x00010001`.

PARAMETERS:

`p_vm`: pointer to the location where the resulting VM structure will be placed.

`p_env`: pointer to the location where the JNI interface pointer for the main thread will be placed.

`vm_args`: Java VM initialization arguments.

RETURNS:

Returns “0” on success; returns a negative number on failure.

DestroyJavaVM

```
jint DestroyJavaVM(JavaVM *vm);
```

Unloads a Java VM and reclaims its resources. Only the main thread can unload the VM. The system waits until the main thread is only remaining user thread before it destroys the VM.

PARAMETERS:

`vm`: the Java VM that will be destroyed.

RETURNS:

Returns “0” on success; returns a negative number on failure.

JDK 1.1.2 does not support unloading the VM.

AttachCurrentThread

```
jint AttachCurrentThread(JavaVM *vm, JNIEnv **p_env,
     void *thr_args);
```

Attaches the current thread to a Java VM. Returns a JNI interface pointer in the `JNIEnv` argument.

Trying to attach a thread that is already attached is a no-op.

A native thread cannot be attached simultaneously to two Java VMs.

---

* JDK 1.1 did not require the native code to set the version field. For backward compatibility, JDK 1.1.2 assumes that the requested version is 0x00010001 if the version field is not set. Future versions of JDK will require the version field to be set to an appropriate value.

PARAMETERS:

`vm`: the VM to which the current thread will be attached.

`p_env`: pointer to the location where the JNI interface pointer of the current thread will be placed.

`thr_args`: VM-specific thread attachment arguments.

RETURNS:

Returns “0” on success; returns a negative number on failure.

DetachCurrentThread

```
jint DetachCurrentThread(JavaVM *vm);
```

Detaches the current thread from a Java VM. All Java monitors held by this thread are released. All Java threads waiting for this thread to die are notified.

The main thread, which is the thread that created the Java VM, cannot be detached from the VM. Instead, the main thread must call `JNI_DestroyJavaVM()` to unload the entire VM.

PARAMETERS:

`vm`: the VM from which the current thread will be detached.

RETURNS:

Returns “0” on success; returns a negative number on failure.

JAVASOFT
A Sun Microsystems, Inc. Business

2550 Garcia Avenue
Mountain View, CA 94043
408-343-1400

For U.S. Sales Office locations, call:
800 821-4643
In California:
800 821-4642

Australia: (02) 844 5000
Belgium: 32 2 716 7911
Canada: 416 477-6745
Finland: +358-0-525561
France: (1) 30 67 50 00
Germany: (0) 89-46 00 8-0
Hong Kong: 852 802 4188
Italy: 039 60551
Japan: (03) 5717-5000
Korea: 822-563-8700
Latin America: 415 688-9464
The Netherlands: 033 501234
New Zealand: (04) 499 2344
Nordic Countries: +46 (0) 8 623 90 00
PRC: 861-849 2828
Singapore: 224 3388
Spain: (91) 5551648
Switzerland: (1) 825 71 11
Taiwan: 2-514-0567
UK: 0276 20444

Elsewhere in the world,
call Corporate Headquarters:
415 960-1300
Intercontinental Sales: 415 688-9000