JNI Functions

Chapter 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:

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,

    FromReflectedMethod,
    FromReflectedField,
    ToReflectedMethod,

    GetSuperclass,
    IsAssignableFrom,

    ToReflectedField,

    Throw,
    ThrowNew,
    ExceptionOccurred,
    ExceptionDescribe,
    ExceptionClear,
    FatalError,

    PushLocalFrame,
    PopLocalFrame,

    NewGlobalRef,
    DeleteGlobalRef,
    DeleteLocalRef,
    IsSameObject,
    NewLocalRef,
    EnsureLocalCapacity,

    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,
    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,
    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,
    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,

    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,

    RegisterNatives,
    UnregisterNatives,
    MonitorEnter,
    MonitorExit,

    GetJavaVM,

    GetStringRegion,
    GetStringUTFRegion,

    GetPrimitiveArrayCritical,
    ReleasePrimitiveArrayCritical,

    GetStringCritical,
    ReleaseStringCritical,

    NewWeakGlobalRef,
    DeleteWeakGlobalRef,

    ExceptionCheck,

    NewDirectByteBuffer,
    GetDirectBufferAddress,
    GetDirectBufferCapacity,

    GetObjectRefType
  };
```

Version Information

GetVersion

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

Returns the version of the native method interface.

LINKAGE:
Index 4 in the JNIEnv interface function table.

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 JDK/JRE 1.1, `GetVersion()` returns `0x00010001`.

In JDK/JRE 1.2, `GetVersion()` returns `0x00010002`.

In JDK/JRE 1.4, `GetVersion()` returns `0x00010004`.

In JDK/JRE 1.6, `GetVersion()` returns `0x00010006`.

Constants

SINCE JDK/JRE 1.2:

```
#define JNI_VERSION_1_1 0x00010001
#define JNI_VERSION_1_2 0x00010002

/* Error codes */
#define JNI_EDETACHED    (-2)              /* thread detached from the VM */
#define JNI_EVERSION     (-3)              /* JNI version error
```

SINCE JDK/JRE 1.4:

```
    #define JNI_VERSION_1_4 0x00010004
```

SINCE JDK/JRE 1.6:

```
    #define JNI_VERSION_1_6 0x00010006
```

Class Operations

DefineClass

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

Loads a class from a buffer of raw class data. The buffer containing the raw class data is not referenced by the VM after the DefineClass call returns, and it may be discarded if desired.

LINKAGE:

Index 5 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`name`: the name of the class or interface to be defined. The string is encoded in modified UTF-8.

`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.

`SecurityException`: if the caller attempts to define a class in the "java" package tree.

FindClass

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

In JDK release 1.1, 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.

Since Java 2 SDK release 1.2, the Java security model allows non-system classes to load and call native methods. `FindClass` locates the class loader associated with the current native method; that is, the class loader of the class that declared the native method. If the native method belongs to a system class, no class loader will be involved. Otherwise, the proper class loader will be invoked to load and link the named class.

Since Java 2 SDK release 1.2, when `FindClass` is called through the Invocation Interface, there is no current native method or its associated class loader. In that case, the result of `ClassLoader.getSystemClassLoader` is used. This is the class loader the virtual machine creates for applications, and is able to locate classes listed in the `java.class.path` property.

The `name` argument is a fully-qualified class name or an array type signature . For example, the fully-qualified class name for the `java.lang.String` class is:

```
                   "java/lang/String"
```

The array type signature of the array class `java.lang.Object[]` is:

```
                   "[Ljava/lang/Object;"
```

LINKAGE:

Index 6 in the JNIEnv interface function table.

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. The string is encoded in modified UTF-8.

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`.

LINKAGE:

Index 10 in the JNIEnv interface function table.

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`.

LINKAGE:

Index 11 in the JNIEnv interface function table.

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.

LINKAGE:
Index 13 in the JNIEnv interface function table.

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.

LINKAGE:

Index 14 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

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

`message`: the message used to construct the `java.lang.Throwable` object. The string is encoded in modified UTF-8.

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.

LINKAGE:

Index 15 in the JNIEnv interface function table.

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.

LINKAGE:

Index 16 in the JNIEnv interface function table.

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.

LINKAGE:

Index 17 in the JNIEnv interface function table.

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.

LINKAGE:

Index 18 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`msg`: an error message. The string is encoded in modified UTF-8.

ExceptionCheck

We introduce a convenience function to check for pending exceptions without creating a local reference to the exception object.

```
jboolean ExceptionCheck(JNIEnv *env);
```

Returns `JNI_TRUE` when there is a pending exception; otherwise, returns `JNI_FALSE`.

LINKAGE:

Index 228 in the JNIEnv interface function table.

SINCE:

JDK/JRE 1.2

Global and Local References

Global 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()`.

LINKAGE:

Index 21 in the JNIEnv interface function table.

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`.

LINKAGE:

Index 22 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`globalRef`: a global reference.

Local References

Local references are valid for the duration of a native method call. They are freed automatically after the native method returns. Each local reference costs some amount of Java Virtual Machine resource. Programmers need to make sure that native methods do not excessively allocate local references. Although local references are automatically freed after the native method returns to Java, excessive allocation of local references may cause the VM to run out of memory during the execution of a native method.

DeleteLocalRef

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

Deletes the local reference pointed to by `localRef`.

LINKAGE:

Index 23 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`localRef`: a local reference.

Note

JDK/JRE 1.1 provides the `DeleteLocalRef` function above so that programmers can manually delete local references. For example, if native code iterates through a potentially large array of objects and uses one element in each iteration, it is a good practice to delete the local reference to the no-longer-used array element before a new local reference is created in the next iteration.

As of JDK/JRE 1.2 an additional set of functions are provided for local reference lifetime management. They are the four functions listed below.

EnsureLocalCapacity

```
jint EnsureLocalCapacity(JNIEnv *env, jint capacity);
```

Ensures that *at least* a given number of local references can be created in the current thread. Returns 0 on success; otherwise returns a negative number and throws an `OutOfMemoryError`.

Before it enters a native method, the VM automatically ensures that at least **16** local references can be created.

For backward compatibility, the VM allocates local references beyond the ensured capacity. (As a debugging support, the VM may give the user warnings that too many local references are being created. In the JDK, the programmer can supply the `-verbose:jni` command line option to turn on these messages.) The VM calls `FatalError` if no more local references can be created beyond the ensured capacity.

LINKAGE:
Index 26 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

PushLocalFrame

```
jint PushLocalFrame(JNIEnv *env, jint capacity);
```

Creates a new local reference frame, in which at least a given number of local references can be created. Returns 0 on success, a negative number and a pending `OutOfMemoryError` on failure.

Note that local references already created in previous local frames are still valid in the current local frame.

LINKAGE:
Index 19 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

PopLocalFrame

```
jobject PopLocalFrame(JNIEnv *env, jobject result);
```

Pops off the current local reference frame, frees all the local references, and returns a local reference in the previous local reference frame for the given `result` object.

Pass `NULL` as `result` if you do not need to return a reference to the previous frame.

LINKAGE:
Index 20 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

NewLocalRef

```
jobject NewLocalRef(JNIEnv *env, jobject ref);
```

Creates a new local reference that refers to the same object as `ref`. The given `ref` may be a global or local reference. Returns `NULL` if `ref` refers to `null`.

LINKAGE:
Index 25 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

Weak Global References

Weak global references are a special kind of global reference. Unlike normal global references, a weak global reference allows the underlying Java object to be garbage collected. Weak global references may be used in any situation where global or local references are used. When the garbage collector runs, it frees the underlying object if the object is only referred to by weak references. A weak global reference pointing to a freed object is functionally equivalent to `NULL`. Programmers can detect whether a weak global reference points to a freed object by using `IsSameObject` to compare the weak reference against `NULL`.

Weak global references in JNI are a simplified version of the Java Weak References, available as part of the Java 2 Platform API (`java.lang.ref` package and its classes).

Clarification *(added June 2001)*

Since garbage collection may occur while native methods are running, objects referred to by weak global references can be freed at any time. While weak global references can be used where global references are used, it is generally inappropriate to do so, as they may become functionally equivalent to `NULL` *without notice.*

While `IsSameObject` *can be used to determine whether a weak global reference refers to a freed object, it does not prevent the object from being freed immediately thereafter. Consequently, programmers may not rely on this check to determine whether a weak global reference may used (as a non-*`NULL` *reference) in any future JNI function call.*

To overcome this inherent limitation, it is recommended that a standard (strong) local or global reference to the same object be acquired using the JNI functions `NewLocalRef` *or* `NewGlobalRef`*, and that this strong reference be used to access the intended object. These functions will return* `NULL` *if the object has been freed, and otherwise will return a strong reference (which will prevent the object from being freed). The new reference should be explicitly deleted when immediate access to the object is no longer required, allowing the object to be freed.*

The weak global reference is weaker than other types of weak references (Java objects of the SoftReference or WeakReference classes). A weak global reference to a specific object will not become functionally equivalent to `NULL` *until after SoftReference or WeakReference objects referring to that same specific object have had their references cleared.*

The weak global reference is weaker than Java's internal references to objects requiring finalization. A weak global reference will not become functionally equivalent to `NULL` *until after the completion of the finalizer for the referenced object, if present.*

Interactions between weak global references and PhantomReferences are undefined. In particular, implementations of a Java VM may (or may not) process weak global references after PhantomReferences, and it may (or may not) be possible to use weak global references to hold on to objects which are also referred to by PhantomReference objects. This undefined use of weak global references should be avoided.

NewWeakGlobalRef

```
jweak NewWeakGlobalRef(JNIEnv *env, jobject obj);
```

Creates a new weak global reference. Returns `NULL` if `obj` refers to `null`, or if the VM runs out of memory. If the VM runs out of memory, an `OutOfMemoryError` will be thrown.

LINKAGE:

Index 226 in the JNIEnv interface function table.

SINCE:

JDK/JRE 1.2

DeleteWeakGlobalRef

`void DeleteWeakGlobalRef(JNIEnv *env, jweak obj);`

Delete the VM resources needed for the given weak global reference.

LINKAGE:
Index 227 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

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.

LINKAGE:
Index 27 in the JNIEnv interface function table.

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, const 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.

LINKAGE:

Index 28 in the JNIEnv interface function table.

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.

LINKAGE:

Index 30 in the JNIEnv interface function table.

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.

LINKAGE:

Index 29 in the JNIEnv interface function table.

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.

LINKAGE:
Index 31 in the JNIEnv interface function table.

PARAMETERS:
`env`: the JNI interface pointer.

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

RETURNS:
Returns a Java class object.

GetObjectRefType

```
jobjectRefType GetObjectRefType(JNIEnv* env, jobject obj);
```

Returns the type of the object referred to by the `obj` argument. The argument `obj` can either be a local, global or weak global reference.

LINKAGE:
Index 232 in the JNIEnv interface function table.

PARAMETERS:
`env`: the JNI interface pointer.

`obj`: a local, global or weak global reference.

`vm`: the virtual machine instance from which the interface will be retrieved.

`env`: pointer to the location where the JNI interface pointer for the current thread will be placed.

`version`: the requested JNI version.

RETURNS:

The function `GetObjectRefType` returns one of the following enumerated values defined as a `jobjectRefType`:

```
JNIInvalidRefType = 0,
JNILocalRefType = 1,
JNIGlobalRefType = 2,
JNIWeakGlobalRefType = 3
```

If the argument `obj` is a weak global reference type, the return will be `JNIWeakGlobalRefType`.

If the argument `obj` is a global reference type, the return value will be `JNIGlobalRefType`.

If the argument `obj` is a local reference type, the return will be `JNILocalRefType`.

If the `obj` argument is not a valid reference, the return value for this function will be `JNIInvalidRefType`.

An invalid reference is a reference which is not a valid handle. That is, the `obj` pointer address does not point to a location in memory which has been allocated from one of the Ref creation functions or returned from a JNI function.

As such, `NULL` would be an invalid reference and `GetObjectRefType(env,NULL)` would return `JNIInvalidRefType`.

On the other hand, a null reference, which is a reference that points to a null, would return the type of reference that the null reference was originally created as.

`GetObjectRefType` cannot be used on deleted references.

Since references are typically implemented as pointers to memory data structures that can potentially be reused by any of the reference allocation services in the VM, once deleted, it is not specified what value the `GetObjectRefType` will return.

SINCE:

JDK/JRE 1.6

IsInstanceOf

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

Tests whether an object is an instance of a class.

LINKAGE:
Index 32 in the JNIEnv interface function table.

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.

LINKAGE:
Index 24 in the JNIEnv interface function table.

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.

LINKAGE:

Index 94 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

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

`sig`: the field signature in a 0-terminated modified 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 4-1a 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

LINKAGE:

Indices in the JNIEnv interface function table:
Table 4-1b Get<type>Field Family of Accessor Routines

Get<type>Field Routine Name	**Index**
`GetObjectField()`	95
`GetBooleanField()`	96

`GetByteField()`	97
`GetCharField()`	98
`GetShortField()`	99
`GetIntField()`	100
`GetLongField()`	101
`GetFloatField()`	102
`GetDoubleField()`	103

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 4-2a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

Set<type>Field Routine	Index
`SetObjectField()`	104
`SetBooleanField()`	105
`SetByteField()`	106
`SetCharField()`	107
`SetShortField()`	108

`SetIntField()`	109
`SetLongField()`	110
`SetFloatField()`	111
`SetDoubleField()`	112

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.

LINKAGE:

Index 33 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

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

`sig`: the method signature in 0-terminated modified 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, const 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 4-3a 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

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

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

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

LINKAGE:

Indices in the JNIEnv interface function table:

Table 4-3b Instance Method Calling Routines

Call<type>Method Routine Name	**Index**

`CallVoidMethod()`
`CallVoidMethodA()`
`CallVoidMethodV()`
	61
	63
	62

`CallObjectMethod()`	34
`CallObjectMethodA()`	36
`CallObjectMethodV()`	35
`CallBooleanMethod()`	37
`CallBooleanMethodA()`	39
`CallBooleanMethodV()`	38
`CallByteMethod()`	40
`CallByteMethodA()`	42
`CallByteMethodV()`	41
`CallCharMethod()`	43
`CallCharMethodA()`	45
`CallCharMethodV()`	44
`CallShortMethod()`	46
`CallShortMethodA()`	48
`CallShortMethodV()`	47
`CallIntMethod()`	49
`CallIntMethodA()`	51
`CallIntMethodV()`	50
`CallLongMethod()`	52
`CallLongMethodA()`	54
`CallLongMethodV()`	53
`CallFloatMethod()`	55

`CallFloatMethodA()`	57
`CallFloatMethodV()`	56
`CallDoubleMethod()`	58
`CallDoubleMethodA()`	60
`CallDoubleMethodV()`	59

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, const 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 4-4a 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

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

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

`CallNonvirtualDoubleMethod()`
`CallNonvirtualDoubleMethodA()`
	jdouble

`CallNonvirtualDoubleMethodV()`

LINKAGE:

Indices in the JNIEnv interface function table.

Table 4-4b CallNonvirtual<type>Method Routines

CallNonvirtual<type>Method Routine Name	Index
`CallNonvirtualVoidMethod()`	91
`CallNonvirtualVoidMethodA()`	93
`CallNonvirtualVoidMethodV()`	92
`CallNonvirtualObjectMethod()`	64
`CallNonvirtualObjectMethodA()`	66
`CallNonvirtualObjectMethodV()`	65
`CallNonvirtualBooleanMethod()`	67
`CallNonvirtualBooleanMethodA()`	69
`CallNonvirtualBooleanMethodV()`	68
`CallNonvirtualByteMethod()`	70
`CallNonvirtualByteMethodA()`	72
`CallNonvirtualByteMethodV()`	71
`CallNonvirtualCharMethod()`	73
`CallNonvirtualCharMethodA()`	75
`CallNonvirtualCharMethodV()`	74
`CallNonvirtualShortMethod()`	76
`CallNonvirtualShortMethodA()`	78
`CallNonvirtualShortMethodV()`	77
`CallNonvirtualIntMethod()`	79
`CallNonvirtualIntMethodA()`	81
`CallNonvirtualIntMethodV()`	80
`CallNonvirtualLongMethod()`	82

`CallNonvirtualLongMethodA()`	84
`CallNonvirtualLongMethodV()`	83
`CallNonvirtualFloatMethod()`	85
`CallNonvirtualFloatMethodA()`	87
`CallNonvirtualFloatMethodV()`	86
`CallNonvirtualDoubleMethod()`	88
`CallNonvirtualDoubleMethodA()`	90
`CallNonvirtualDoubleMethodV()`	89

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.

LINKAGE:

Index 144 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

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

`sig`: the field signature in a 0-terminated modified 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 4-5a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

GetStatic<type>Field Routine Name	Index

`GetStaticObjectField()`	145
`GetStaticBooleanField()`	146
`GetStaticByteField()`	147
`GetStaticCharField()`	148
`GetStaticShortField()`	149
`GetStaticIntField()`	150
`GetStaticLongField()`	151
`GetStaticFloatField()`	152
`GetStaticDoubleField()`	153

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 4-6a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

SetStatic<type>Field Routine Name	**Index**
`SetStaticObjectField()`	154
`SetStaticBooleanField()`	155
`SetStaticByteField()`	156

`SetStaticCharField()`	157
`SetStaticShortField()`	158
`SetStaticIntField()`	159
`SetStaticLongField()`	160
`SetStaticFloatField()`	161
`SetStaticDoubleField()`	162

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.

LINKAGE:
Index 113 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`clazz`: a Java class object.

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

`sig`: the method signature in a 0-terminated modified 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 4-7a 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

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

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

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

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

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

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

LINKAGE:

Indices in the JNIEnv interface function table.

Table 4-7b CallStatic<type>Method Calling Routines

CallStatic<type>Method Routine Name	**Index**
`CallStaticVoidMethod()`	141
`CallStaticVoidMethodA()`	143
`CallStaticVoidMethodV()`	142
`CallStaticObjectMethod()`	114
`CallStaticObjectMethodA()`	116
`CallStaticObjectMethodV()`	115
`CallStaticBooleanMethod()`	117
`CallStaticBooleanMethodA()`	119

`CallStaticBooleanMethodV()`	118
`CallStaticByteMethod()`	120
`CallStaticByteMethodA()`	122
`CallStaticByteMethodV()`	121
`CallStaticCharMethod()`	123
`CallStaticCharMethodA()`	125
`CallStaticCharMethodV()`	124
`CallStaticShortMethod()`	126
`CallStaticShortMethodA()`	128
`CallStaticShortMethodV()`	127
`CallStaticIntMethod()`	129
`CallStaticIntMethodA()`	131
`CallStaticIntMethodV()`	130
`CallStaticLongMethod()`	132
`CallStaticLongMethodA()`	134
`CallStaticLongMethodV()`	133
`CallStaticFloatMethod()`	135
`CallStaticFloatMethodA()`	137
`CallStaticFloatMethodV()`	136
`CallStaticDoubleMethod()`	138
`CallStaticDoubleMethodA()`	140
`CallStaticDoubleMethodV()`	139

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.

LINKAGE:

Index 163 in the JNIEnv interface function table.

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.

LINKAGE:

Index 164 in the JNIEnv interface function table.

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.

LINKAGE:

Index 165 in the JNIEnv interface function table.

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()`.

LINKAGE:
Index 166 in the JNIEnv interface function table.

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 characters in modified UTF-8 encoding.

LINKAGE:
Index 167 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`bytes`: the pointer to a modified 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 length in bytes of the modified UTF-8 representation of a string.

LINKAGE:
Index 168 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

RETURNS:

Returns the UTF-8 length of the string.

GetStringUTFChars

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

Returns a pointer to an array of bytes representing the string in modified UTF-8 encoding. 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.

LINKAGE:
Index 169 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`string`: a Java string object.

`isCopy`: a pointer to a boolean.

RETURNS:
Returns a pointer to a modified 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()`.

LINKAGE:
Index 170 in the JNIEnv interface function table.

PARAMETERS:
`env`: the JNI interface pointer.

`string`: a Java string object.

`utf`: a pointer to a modified UTF-8 string.

Note
In JDK/JRE 1.1, programmers can get primitive array elements in a user-supplied buffer. As of JDK/JRE 1.2 additional set of functions are provided allowing native code to obtain characters in Unicode (UTF-16) or modified UTF-8 encoding in a user-supplied buffer. See the functions below.

GetStringRegion

```
void GetStringRegion(JNIEnv *env, jstring str, jsize start, jsize len, jchar *buf);
```

Copies `len` number of Unicode characters beginning at offset `start` to the given buffer `buf`.

Throws `StringIndexOutOfBoundsException` on index overflow.

LINKAGE:
Index 220 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

GetStringUTFRegion

< `void GetStringUTFRegion(JNIEnv *env, jstring str, jsize start, jsize len, char *buf);`

Translates `len` number of Unicode characters beginning at offset `start` into modified UTF-8 encoding and place the result in the given buffer `buf`.

Throws `StringIndexOutOfBoundsException` on index overflow.

LINKAGE:
Index 221 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

GetStringCritical
ReleaseStringCritical

```
const jchar * GetStringCritical(JNIEnv *env, jstring string, jboolean *isCopy);
void ReleaseStringCritical(JNIEnv *env, jstring string, const jchar *carray);
```

The semantics of these two functions are similar to the existing `Get/ReleaseStringChars` functions. If possible, the VM returns a pointer to string elements; otherwise, a copy is made. **However, there are significant restrictions on how these functions can be used.** In a code segment enclosed by `Get/ReleaseStringCritical` calls, the native code must not issue arbitrary JNI calls, or cause the current thread to block.

The restrictions on `Get/ReleaseStringCritical` are similar to those on `Get/ReleasePrimitiveArrayCritical`.

LINKAGE (GetStringCritical):
Index 224 in the JNIEnv interface function table.

LINKAGE (ReleaseStingCritical):
Index 225 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

Array Operations

GetArrayLength

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

Returns the number of elements in the array.

LINKAGE:

Index 171 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array object.

RETURNS:

Returns the length of the array.

NewObjectArray

```
jobjectArray 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`.

LINKAGE:

Index 172 in the JNIEnv interface function table.

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.

LINKAGE:
Index 173 in the JNIEnv interface function table.

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.

LINKAGE:
Index 174 in the JNIEnv interface function table.

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 4-8a 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

LINKAGE:
Indices in the JNIEnv interface function table.

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

New<PrimitiveType>Array Routines	Index
`NewBooleanArray()`	175
`NewByteArray()`	176
`NewCharArray()`	177
`NewShortArray()`	178
`NewIntArray()`	179
`NewLongArray()`	180
`NewFloatArray()`	181
`NewDoubleArray()`	182

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 4-9a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

Get<PrimitiveType>ArrayElements Routines	**Index**
`GetBooleanArrayElements()`	183
`GetByteArrayElements()`	184
`GetCharArrayElements()`	185
`GetShortArrayElements()`	186
`GetIntArrayElements()`	187
`GetLongArrayElements()`	188
`GetFloatArrayElements()`	189
`GetDoubleArrayElements()`	190

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 4-11a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

Release<PrimitiveType>ArrayElements Routines	**Index**
`ReleaseBooleanArrayElements()`	191
`ReleaseByteArrayElements()`	192
`ReleaseCharArrayElements()`	193
`ReleaseShortArrayElements()`	194
`ReleaseIntArrayElements()`	195
`ReleaseLongArrayElements()`	196
`ReleaseFloatArrayElements()`	197
`ReleaseDoubleArrayElements()`	198

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 4-12a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

Get<PrimitiveType>ArrayRegion Routine	**Index**
`GetBooleanArrayRegion()`	199
`GetByteArrayRegion()`	200
`GetCharArrayRegion()`	201
`GetShortArrayRegion()`	202
`GetIntArrayRegion()`	203
`GetLongArrayRegion()`	204
`GetFloatArrayRegion()`	205
`GetDoubleArrayRegion()`	206

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,` const *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 4-13a 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

LINKAGE:

Indices in the JNIEnv interface function table.

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

Set<PrimitiveType>ArrayRegion Routine	**Index**
`SetBooleanArrayRegion()`	207
`SetByteArrayRegion()`	208
`SetCharArrayRegion()`	209
`SetShortArrayRegion()`	210
`SetIntArrayRegion()`	211
`SetLongArrayRegion()`	212
`SetFloatArrayRegion()`	213
`SetDoubleArrayRegion()`	214

PARAMETERS:

`env`: the JNI interface pointer.

`array`: a Java array.

`start`: the starting index.

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

`buf`: the source buffer.

THROWS:

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

> **Note**
>
> As of JDK/JRE 1.1, programmers can use `Get/Release<primitivetype>ArrayElements` functions to obtain a pointer to primitive array elements. If the VM supports pinning, the pointer to the original data is returned; otherwise, a copy is made.
>
> New functions introduced as of JDK/JRE 1.3 allow native code to obtain a direct pointer to array elements even if the VM does not support pinning.

GetPrimitiveArrayCritical ReleasePrimitiveArrayCritical

```
void * GetPrimitiveArrayCritical(JNIEnv *env, jarray array, jboolean *isCopy);
void ReleasePrimitiveArrayCritical(JNIEnv *env, jarray array, void *carray, jint mode);
```

The semantics of these two functions are very similar to the existing `Get/Release<primitivetype>ArrayElements functions. If possible, the VM returns a pointer to the primitive array; otherwise, a copy is made.` **`However, there are significant restrictions on how these functions can be used.`**

After calling `GetPrimitiveArrayCritical`, the native code should not run for an extended period of time before it calls `ReleasePrimitiveArrayCritical`. We must treat the code inside this pair of functions as running in a "critical region." Inside a critical region, native code must not call other JNI functions, or any system call that may cause the current thread to block and wait for another Java thread. (For example, the current thread must not call `read` on a stream being written by another Java thread.)

These restrictions make it more likely that the native code will obtain an uncopied version of the array, even if the VM does not support pinning. For example, a VM may temporarily disable garbage collection when the native code is holding a pointer to an array obtained via `GetPrimitiveArrayCritical`.

Multiple pairs of `GetPrimtiveArrayCritical` and `ReleasePrimitiveArrayCritical` may be nested. For example:

```
  jint len = (*env)->GetArrayLength(env, arr1);
  jbyte *a1 = (*env)->GetPrimitiveArrayCritical(env, arr1, 0);
  jbyte *a2 = (*env)->GetPrimitiveArrayCritical(env, arr2, 0);
  /* We need to check in case the VM tried to make a copy. */
  if (a1 == NULL || a2 == NULL) {
    ... /* out of memory exception thrown */
  }
  memcpy(a1, a2, len);
  (*env)->ReleasePrimitiveArrayCritical(env, arr2, a2, 0);
  (*env)->ReleasePrimitiveArrayCritical(env, arr1, a1, 0);
```

Note that `GetPrimitiveArrayCritical` might still make a copy of the array if the VM internally represents arrays in a different format. Therefore we need to check its return value against `NULL` for possible out of memory situations.

LINKAGE (GetPrimitiveArrayCritical):

Linkage Index 222 in the JNIEnv interface function table.

LINKAGE (ReleasePrimitiveArrayCritical):

Linkage Index 223 in the JNIEnv interface function table.

SINCE:

JDK/JRE 1.2

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 `name` and `signature` fields of the JNINativeMethod structure are pointers to modified UTF-8 strings. 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, ...);
```

LINKAGE:

Index 215 in the JNIEnv interface function table.

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.

LINKAGE:

Index 216 in the JNIEnv interface function table.

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`.

Enters the monitor associated with the object referred to by obj. The `obj` reference must not be `NULL`.

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.

A monitor entered through a `MonitorEnter` JNI function call cannot be exited using the `monitorexit` Java virtual machine instruction or a synchronized method return. A `MonitorEnter` JNI function call and a `monitorenter` Java virtual machine instruction may race to enter the monitor associated with the same object.

To avoid deadlocks, a monitor entered through a `MonitorEnter` JNI function call must be exited using the `MonitorExit` JNI call, unless the `DetachCurrentThread` call is used to implicitly release JNI monitors.

LINKAGE:

Index 217 in the JNIEnv interface function table.

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.

Native code must not use `MonitorExit` to exit a monitor entered through a synchronized method or a `monitorenter` Java virtual machine instruction.

LINKAGE:
Index 218 in the JNIEnv interface function table.

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.

EXCEPTIONS:
`IllegalMonitorStateException`: if the current thread does not own the monitor.

NIO Support

The NIO-related entry points allow native code to access `java.nio` *direct buffers*. The contents of a direct buffer can, potentially, reside in native memory outside of the ordinary garbage-collected heap. For information about direct buffers, please see New I/O APIs and the specification of the `java.nio.ByteBuffer` class.

Three new functions introduced in JDK/JRE 1.4 allow JNI code to create, examine, and manipulate direct buffers:

- `NewDirectByteBuffer`
- `GetDirectBufferAddress`

- GetDirectBufferCapacity

Every implementation of the Java virtual machine must support these functions, but not every implementation is required to support JNI access to direct buffers. If a JVM does not support such access then the `NewDirectByteBuffer` and `GetDirectBufferAddress` functions must always return `NULL`, and the `GetDirectBufferCapacity` function must always return `-1`. If a JVM *does* support such access then these three functions must be implemented to return the appropriate values.

NewDirectByteBuffer

```
jobject NewDirectByteBuffer(JNIEnv* env, void* address, jlong capacity);
```

Allocates and returns a direct `java.nio.ByteBuffer` referring to the block of memory starting at the memory address `address` and extending `capacity` bytes.

Native code that calls this function and returns the resulting byte-buffer object to Java-level code should ensure that the buffer refers to a valid region of memory that is accessible for reading and, if appropriate, writing. An attempt to access an invalid memory location from Java code will either return an arbitrary value, have no visible effect, or cause an unspecified exception to be thrown.

LINKAGE:

Index 229 in the JNIEnv interface function table.

PARAMETERS:

`env`: the `JNIEnv` interface pointer

`address`: the starting address of the memory region (must not be `NULL`)

`capacity`: the size in bytes of the memory region (must be positive)

RETURNS:

Returns a local reference to the newly-instantiated `java.nio.ByteBuffer` object. Returns `NULL` if an exception occurs, or if JNI access to direct buffers is not supported by this virtual machine.

EXCEPTIONS:

`OutOfMemoryError`: if allocation of the `ByteBuffer` object fails

SINCE:
JDK/JRE 1.4

GetDirectBufferAddress

`void* GetDirectBufferAddress(JNIEnv* env, jobject buf);`

Fetches and returns the starting address of the memory region referenced by the given direct `java.nio.Buffer`.

This function allows native code to access the same memory region that is accessible to Java code via the buffer object.

LINKAGE:
Index 230 in the JNIEnv interface function table.

PARAMETERS:

`env`: the `JNIEnv` interface pointer

`buf`: a direct `java.nio.Buffer` object (must not be `NULL`)

RETURNS:

Returns the starting address of the memory region referenced by the buffer. Returns `NULL` if the memory region is undefined, if the given object is not a direct `java.nio.Buffer`, or if JNI access to direct buffers is not supported by this virtual machine.

SINCE:
JDK/JRE 1.4

GetDirectBufferCapacity

`jlong GetDirectBufferCapacity(JNIEnv* env, jobject buf);`

Fetches and returns the capacity of the memory region referenced by the given direct `java.nio.Buffer`. The capacity is the number of *elements* that the memory region contains.

LINKAGE:
Index 231 in the JNIEnv interface function table.

PARAMETERS:

`env`: the `JNIEnv` interface pointer

`buf`: a direct `java.nio.Buffer` object (must not be `NULL`)

RETURNS:

Returns the capacity of the memory region associated with the buffer. Returns `-1` if the given object is not a direct `java.nio.Buffer`, if the object is an unaligned view buffer and the processor architecture does not support unaligned access, or if JNI access to direct buffers is not supported by this virtual machine.

SINCE:

JDK/JRE 1.4

Reflection Support

Programmers can use the JNI to call Java methods or access Java fields if they know the name and type of the methods or fields. The Java Core Reflection API allows programmers to introspect Java classes at runtime. JNI provides a set of conversion functions between field and method IDs used in the JNI to field and method objects used in the Java Core Reflection API.

FromReflectedMethod

```
jmethodID FromReflectedMethod(JNIEnv *env, jobject method);
```

Converts a `java.lang.reflect.Method` or `java.lang.reflect.Constructor` object to a method ID.

LINKAGE:

Index 7 in the JNIEnv interface function table.

SINCE:

JDK/JRE 1.2

FromReflectedField

```
jfieldID FromReflectedField(JNIEnv *env, jobject field);
```

Converts a `java.lang.reflect.Field` to a field ID.

LINKAGE:
Index 8 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

ToReflectedMethod

```
jobject ToReflectedMethod(JNIEnv *env, jclass cls,
   jmethodID methodID, jboolean isStatic);
```

Converts a method ID derived from `cls` to a `java.lang.reflect.Method` or `java.lang.reflect.Constructor` object. `isStatic` must be set to `JNI_TRUE` if the method ID refers to a static field, and `JNI_FALSE` otherwise.

Throws `OutOfMemoryError` and returns 0 if fails.

LINKAGE:
Index 9 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

ToReflectedField

```
jobject ToReflectedField(JNIEnv *env, jclass cls,
   jfieldID fieldID, jboolean isStatic);
```

Converts a field ID derived from `cls` to a `java.lang.reflect.Field` object. `isStatic` must be set to `JNI_TRUE` if `fieldID` refers to a static field, and `JNI_FALSE` otherwise.

Throws `OutOfMemoryError` and returns 0 if fails.

LINKAGE:
Index 12 in the JNIEnv interface function table.

SINCE:
JDK/JRE 1.2

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`.

LINKAGE:

Index 219 in the JNIEnv interface function table.

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.