Chapter 4: JNI Functions

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. Passing NULL when explicity not allowed may result in an unexpected exception or a fatal crash.

Functions whose definition may both return `NULL` and throw an exception on error, may choose only to return `NULL` to indicate an error, but not throw any exception. For example, a JNI implementation may consider an "out of memory" condition temporary, and may not wish to throw an `OutOfMemoryError` since this would appear fatal (JDK API `java.lang.Error` documentation: "*indicates serious problems that a reasonable application should not try to catch*").

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 the following code example. 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.

```
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,

    GetModule,

    IsVirtualThread
  };
```

Constants

There are a number of general constants used throughout the JNI API.

Boolean values

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

Return codes

General return value constants for JNI functions.

```
#define JNI_OK          0               /* success */
#define JNI_ERR         (-1)            /* unknown error */
#define JNI_EDETACHED   (-2)            /* thread detached from the VM */
#define JNI_EVERSION    (-3)            /* JNI version error */
#define JNI_ENOMEM      (-4)            /* not enough memory */
#define JNI_EEXIST      (-5)            /* VM already created */
#define JNI_EINVAL      (-6)            /* invalid arguments */
```

Version Information

GetVersion

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

Returns the version of the native method interface. For Java SE Platform 19 and later, it returns `JNI_VERSION_19`. The following table gives the version of JNI included in each release of the Java SE Platform (for older versions of JNI, the JDK release is used instead of the Java SE Platform):

Java SE Platform	JNI Version
1.1	JNI_VERSION_1_1
1.2	JNI_VERSION_1_2
1.3	JNI_VERSION_1_2
1.4	JNI_VERSION_1_4
5.0	JNI_VERSION_1_4
6	JNI_VERSION_1_6
7	JNI_VERSION_1_6
8	JNI_VERSION_1_8
9	JNI_VERSION_9
10	JNI_VERSION_10
11	JNI_VERSION_10

Java SE Platform	JNI Version
12	JNI_VERSION_10
13	JNI_VERSION_10
14	JNI_VERSION_10
15	JNI_VERSION_10
16	JNI_VERSION_10
17	JNI_VERSION_10
18	JNI_VERSION_10
19+	JNI_VERSION_19

LINKAGE:

Index 4 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

RETURNS:

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

Version Constants

```
#define JNI_VERSION_1_1 0x00010001
#define JNI_VERSION_1_2 0x00010002
#define JNI_VERSION_1_4 0x00010004
#define JNI_VERSION_1_6 0x00010006
#define JNI_VERSION_1_8 0x00010008
#define JNI_VERSION_9   0x00090000
#define JNI_VERSION_10  0x000a0000
#define JNI_VERSION_19  0x00130000
```

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, must not be NULL.

name: the name of the class or interface to be defined. The string is encoded in modified UTF-8. This value may be NULL, or it must match the name encoded within the class file data.

loader: a class loader assigned to the defined class. This value may be NULL, indicating the "*null class loader*" (or "*bootstrap class loader*").

buf: buffer containing the .class file data. A NULL value will cause a ClassFormatError.

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 JDK 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, link, and initialize, the named class.

Since JDK 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.

If FindClass is called from a library lifecycle function hook, the class loader is determined as follows:

- for JNI_OnLoad and JNI_OnLoad_L the class loader of the class that is loading the native library is used
- for JNI_OnUnload and JNI_OnUnload_L the class loader returned by ClassLoader.getSystemClassLoader is used (as the class loader used at on-load time may no longer exist)

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, must not be NULL.

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. A NULL value may cause NoClassDefFoundError to occur, or a crash.

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, must not be `NULL`.

`clazz`: a Java class object, must not be `NULL`.

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, must not be `NULL`.

`clazz1`: the first class argument, must not be `NULL`.

`clazz2`: the second class argument, must not be `NULL`.

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.

Module Operations

GetModule

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

Returns the `java.lang.Module` object for the module that the class is a member of. If the class is not in a named module then the unnamed module of the class loader for the class is returned. If the class represents an array type then this function returns the `Module` object for the element type. If the class represents a primitive type or `void`, then the `Module` object for the `java.base` module is returned.

LINKAGE:

Index 233 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

clazz: a Java class object, must not be NULL.

RETURNS:

Returns the module that the class or interface is a member of.

SINCE:

JDK/JRE 9

Thread Operations

IsVirtualThread

```
jboolean IsVirtualThread(JNIEnv *env, jobject obj);
```

IsVirtualThread is a preview API of the Java platform. *Preview features may be removed in a future release, or upgraded to permanent features of the Java platform.*

Tests whether an object is a virtual Thread.

LINKAGE:

Index 234 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

obj: a Java object, possibly a NULL value.

RETURNS:

Returns JNI_TRUE if the object is a virtual Thread.

SINCE:

JDK/JRE 19

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, must not be NULL.

obj: a java.lang.Throwable object, must not be NULL.

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, must not be `NULL`.

`clazz`: a subclass of `java.lang.Throwable`, must not be `NULL`.

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

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, must not be `NULL`.

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`. The pending exception is cleared as a side-effect of calling this function. This is a convenience routine provided for debugging.

LINKAGE:

Index 16 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

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, must not be NULL.

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, must not be NULL.

msg: an error message. The string is encoded in modified UTF-8. May be a NULL value.

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.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

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, must not be NULL.

obj: a global or local reference. May be a NULL value, in which case this function will return NULL.

RETURNS:

Returns a global reference to the given obj.

May return NULL if:

- obj refers to null

- the system has run out of memory
- `obj` was a weak global reference and has already been garbage collected

---

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, must not be `NULL`.

`globalRef`: a global reference. May be a `NULL` value, in which case this function does nothing.

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, must not be `NULL`.

`localRef`: a local reference. The function does nothing in the case of a `NULL` value passed here.

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.

Some Java Virtual Machine implementations may choose to limit the maximum `capacity`, which may cause the function to return an error (e.g. `JNI_ERR` or `JNI_EINVAL`). The HotSpot JVM implementation, for example,

uses the -XX:+MaxJNILocalCapacity flag (default: 65536).

LINKAGE:

Index 26 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

capacity: the minimum number of required local references. Must be >= 0.

RETURNS:

JNI_OK upon succcess.

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.

As with EnsureLocalCapacity, some Java Virtual Machine implementations may choose to limit the maximum capacity, which may cause the function to return an error.

LINKAGE:

Index 19 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

capacity: the minimum number of required local references. Must be > 0.

RETURNS:

JNI_OK upon succcess.

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.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

result: an object to be passed to the previous local reference frame, may be NULL.

RETURNS:

Returns a local reference in the previous local reference frame for the given result object, or NULL if the given result object was NULL.

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, a local reference or `NULL`. Returns `NULL` if `ref` refers to `null`.

LINKAGE:

Index 25 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

`ref`: a reference to the object for which the function creates a new local reference. May be a `NULL` value.

RETURNS:

Returns a new local reference that refers to the same object as `ref`.

May return `NULL` if:

- `ref` refers to `null`
- the system has run out of memory
- `ref` was a weak global reference and has already been garbage collected

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.

Weak global references are related to Java phantom references (`java.lang.ref.PhantomReference`). A weak global reference to a specific object is treated as a phantom reference referring to that object when determining whether the object is *phantom reachable* (see `java.lang.ref`). Such a weak global reference will become functionally equivalent to `NULL` at the same time as a `PhantomReference` referring to that same object would be cleared by the garbage collector.

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.

`IsSameObject` can be used to compare a weak global reference to a non-`NULL` local or global reference. If the objects are the same, the weak global reference won't become functionally equivalent to `NULL` so long as the other reference has not been deleted.

`IsSameObject` can also be used to compare a weak global reference to `NULL` to determine whether the underlying object has been freed. However, programmers should not rely on this check to determine whether a weak global reference may be used (as a non-`NULL` reference) in any future JNI function call, since an intervening garbage collection could change the weak global reference.

Instead, it is recommended that a (strong) local or global reference to the underlying object be acquired using one of the JNI functions `NewLocalRef` or `NewGlobalRef`. These functions will return `NULL` if the object has been freed. Otherwise, the new reference will prevent the underlying object from being freed. The new reference, if non-`NULL`, can then be used to access the underlying object, and deleted when such access is no longer needed.

NewWeakGlobalRef

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

Creates a new weak global reference. The weak global reference will not prevent garbage collection of the given object. `IsSameObject` may be used to test if the object referred to by the reference has been freed. Returns `NULL` if `obj` refers to `null`, or if `obj` was a weak global reference, 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.

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

`obj`: the object to create a global weak reference for.

RETURNS:

Return a global weak reference to the given `obj`.

May return `NULL` if:

- `obj` refers to `null`
- the system has run out of memory
- `obj` was a weak global reference and has already been garbage collected

THROWS:

`OutOfMemoryError` if the system runs out of memory.

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.

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

`obj`: global weak reference to delete. This function does nothing if passed `NULL`.

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.

Note: The Java Language Specification, "Implementing Finalization" (JLS §12.6.1) states: "An object o is not finalizable until its constructor has invoked the constructor for Object on o and that invocation has completed

successfully". Since AllocObject() does not invoke a constructor, objects created with this function are not eligible for finalization.

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, must not be NULL.

clazz: a reference to a Java class object, must not be NULL.

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, must not be NULL.

`clazz`: a reference to a Java class object, must not be `NULL`.

`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, must not be `NULL`.

`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, or `NULL`.

LINKAGE:

Index 232 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer, must not be `NULL`.

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

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, must not be `NULL`.

`obj`: a Java object, possibly a `NULL` value.

`clazz`: a Java class object, must not be `NULL`.

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, must not be `NULL`.

`ref1`: a Java object, may be `NULL`.

`ref2`: a Java object, may be `NULL`.

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, must not be `NULL`.

`clazz`: a Java class object, must not be `NULL`.

`name`: the field name in a 0-terminated modified UTF-8 string, must not be `NULL`.

`sig`: the field signature in a 0-terminated modified UTF-8 string, must not be `NULL`.

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.

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

Get<type>Field Routine Name	Native Type
GetLongField()	jlong
GetFloatField()	jfloat
GetDoubleField()	jdouble

LINKAGE:

Indices in the JNIEnv interface function table:

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, must not be NULL.

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.

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

Set<type>Field Routine	Native Type
SetLongField()	jlong
SetFloatField()	jfloat
SetDoubleField()	jdouble

LINKAGE:

Indices in the JNIEnv interface function table.

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, must not be NULL.

obj: a Java object, must not be NULL.

fieldID: a valid field ID.

value: the new value of the field.

Calling Instance Methods

When calling methods from native code be mindful of whether those methods may be caller-sensitive.

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 supertypes 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, must not be NULL.

clazz: a Java class object, must not be NULL.

name: the method name in a 0-terminated modified UTF-8 string, must not be NULL.

`sig`: the method signature in 0-terminated modified UTF-8 string, must not be `NULL`.

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.

Instance Method Calling Routines

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

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

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

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

`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:

Instance Method Calling Routines

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

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

`CallObjectMethod()`
`CallObjectMethodA()`
`CallObjectMethodV()`
	34
	36
	35

`CallBooleanMethod()`
`CallBooleanMethodA()`
`CallBooleanMethodV()`
	37
	39
	38

`CallByteMethod()`
`CallByteMethodA()`
`CallByteMethodV()`
	40
	42
	41

`CallCharMethod()`
`CallCharMethodA()`
`CallCharMethodV()`
	43
	45
	44

`CallShortMethod()`
`CallShortMethodA()`
`CallShortMethodV()`
	46
	48
	47

Call<type>Method Routine Name	Index
CallIntMethod() CallIntMethodA() CallIntMethodV()	49 51 50
CallLongMethod() CallLongMethodA() CallLongMethodV()	52 54 53
CallFloatMethod() CallFloatMethodA() CallFloatMethodV()	55 57 56
CallDoubleMethod() CallDoubleMethodA() CallDoubleMethodV()	58 60 59

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

obj: a Java object, must not be NULL.

methodID: a valid 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 or interface 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 supertypes.

CallNonvirtual<type>Method routines are the mechanism for invoking "*default interface methods*" introduced in Java 8.

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.

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

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

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

LINKAGE:

Indices in the JNIEnv interface function table.

CallNonvirtual<type>Method Routines

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

`CallNonvirtualVoidMethod()`
`CallNonvirtualVoidMethodA()`
`CallNonvirtualVoidMethodV()`
	91
	93
	92

`CallNonvirtualObjectMethod()`
`CallNonvirtualObjectMethodA()`
`CallNonvirtualObjectMethodV()`
	64
	66
	65

`CallNonvirtualBooleanMethod()`
`CallNonvirtualBooleanMethodA()`
`CallNonvirtualBooleanMethodV()`
	67
	69
	68

`CallNonvirtualByteMethod()`
`CallNonvirtualByteMethodA()`
`CallNonvirtualByteMethodV()`
	70
	72
	71

`CallNonvirtualCharMethod()`
`CallNonvirtualCharMethodA()`
`CallNonvirtualCharMethodV()`
	73
	75
	74

`CallNonvirtualShortMethod()`
`CallNonvirtualShortMethodA()`
`CallNonvirtualShortMethodV()`
	76
	78
	77

`CallNonvirtualIntMethod()`
`CallNonvirtualIntMethodA()`
`CallNonvirtualIntMethodV()`
	79
	81
	80

`CallNonvirtualLongMethod()`
`CallNonvirtualLongMethodA()`
`CallNonvirtualLongMethodV()`
	82
	84
	83

`CallNonvirtualFloatMethod()`
`CallNonvirtualFloatMethodA()`
`CallNonvirtualFloatMethodV()`
	85
	87
	86

`CallNonvirtualDoubleMethod()`
`CallNonvirtualDoubleMethodA()`
`CallNonvirtualDoubleMethodV()`
	88
	90
	89

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

`clazz`: a Java class, must not be `NULL`.

`obj`: a Java object, must not be `NULL`.

`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, must not be `NULL`.

`clazz`: a Java class object, must not be `NULL`.

`name`: the static field name in a 0-terminated modified UTF-8 string, must not be `NULL`.

`sig`: the field signature in a 0-terminated modified UTF-8 string, must not be `NULL`.

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.

GetStatic<type>Field Family of Accessor Routines

***GetStatic<type>Field* Routine Name**	**Native Type**
`GetStaticObjectField()`	jobject
`GetStaticBooleanField()`	jboolean

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

LINKAGE:

Indices in the JNIEnv interface function table.

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, must not be NULL.

clazz: a Java class object, must not be NULL.

fieldID: a valid 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.

SetStatic<type>Field Family of Accessor Routines

SetStatic<type>Field Routine Name	NativeType
SetStaticObjectField()	jobject
SetStaticBooleanField()	jboolean
SetStaticByteField()	jbyte

SetStatic<type>Field Routine Name	NativeType
SetStaticCharField()	jchar
SetStaticShortField()	jshort
SetStaticIntField()	jint
SetStaticLongField()	jlong
SetStaticFloatField()	jfloat
SetStaticDoubleField()	jdouble

LINKAGE:

Indices in the JNIEnv interface function table.

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, must not be NULL.

clazz: a Java class object, must not be NULL.

fieldID: a valid static field ID.

value: the new value of the field.

Calling Static Methods

When calling methods from native code be mindful of whether those methods may be caller-sensitive.

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, must not be NULL.

clazz: a Java class object, must not be NULL.

name: the static method name in a 0-terminated modified UTF-8 string, must not be NULL.

`sig`: the method signature in a 0-terminated modified UTF-8 string, must not be `NULL`.

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.

CallStatic<type>Method Calling Routines

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

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

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

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

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

`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.

CallStatic<type>Method Calling Routines

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

`CallStaticVoidMethod()`
`CallStaticVoidMethodA()`
`CallStaticVoidMethodV()`
	141
	143
	142

`CallStaticObjectMethod()`
`CallStaticObjectMethodA()`
`CallStaticObjectMethodV()`
	114
	116
	115

`CallStaticBooleanMethod()`
`CallStaticBooleanMethodA()`
`CallStaticBooleanMethodV()`
	117
	119
	118

`CallStaticByteMethod()`
`CallStaticByteMethodA()`
`CallStaticByteMethodV()`
	120
	122
	121

`CallStaticCharMethod()`
`CallStaticCharMethodA()`
`CallStaticCharMethodV()`
	123
	125
	124

`CallStaticShortMethod()`
`CallStaticShortMethodA()`
`CallStaticShortMethodV()`
	126
	128
	127

`CallStaticIntMethod()`
`CallStaticIntMethodA()`
`CallStaticIntMethodV()`
	129
	131
	130

CallStatic<type>Method Routine Name	Index
CallStaticLongMethod() CallStaticLongMethodA() CallStaticLongMethodV()	132 134 133
CallStaticFloatMethod() CallStaticFloatMethodA() CallStaticFloatMethodV()	135 137 136
CallStaticDoubleMethod() CallStaticDoubleMethodA() CallStaticDoubleMethodV()	138 140 139

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

clazz: a Java class object, must not be NULL.

methodID: a valid 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

This specification makes no assumptions on how a JVM represent Java strings internally. Strings returned from these operations:

- GetStringChars()
- GetStringUTFChars()
- GetStringRegion()
- GetStringUTFRegion()
- GetStringCritical()

are therefore not required to be NULL terminated. Programmers are expected to determine buffer capacity requirements via GetStringLength() or GetStringUTFLength().

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, must not be NULL.

unicodeChars: pointer to a Unicode string. May be a NULL value, in which case len must be 0.

len: length of the Unicode string. May be 0.

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, must not be NULL.

string: a Java string object, must not be NULL.

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, must not be NULL.

string: a Java string object, must not be NULL.

isCopy: a pointer to a boolean, may be a NULL value.

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, must not be NULL.

string: a Java string object, must not be NULL.

chars: a pointer to a Unicode string, as previously returned by GetStringChars().

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, must not be NULL.

bytes: the pointer to a modified UTF-8 string, must not be NULL.

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, must not be NULL.

string: a Java string object, must not be NULL.

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, must not be NULL.

string: a Java string object, must not be NULL.

isCopy: a pointer to a boolean, may be a NULL value.

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, must not be `NULL`.

`string`: a Java string object, must not be `NULL`.

`utf`: a pointer to a modified UTF-8 string, previously returned by `GetStringUTFChars()`.

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`.

LINKAGE:

Index 220 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer, must not be `NULL`.

`str`: a Java string object, must not be `NULL`.

`start`: the index of the first unicode character in the string to copy. Must be greater than or equal to zero, and less than string length ("`GetStringLength()`").

`len`: the number of unicode characters to copy. Must be greater than or equal to zero, and "`start + len`" must be less than string length ("`GetStringLength()`").

`buf`: the unicode character buffer into which to copy the string region. Must not be `NULL` if given `len` is > 0.

THROWS:

`StringIndexOutOfBoundsException`: on index overflow.

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`.

The `len` argument specifies the number of *unicode characters*. The resulting number modified UTF-8 encoding characters may be greater than the given `len` argument. `GetStringUTFLength()` may be used to determine the maximum size of the required character buffer.

Since this specification does not require the resulting string copy be NULL terminated, it is advisable to clear the given character buffer (e.g. "`memset()`") before using this function, in order to safely perform `strlen()`.

LINKAGE:

Index 221 in the JNIEnv interface function table.

PARAMETERS:

`env`: the JNI interface pointer, must not be `NULL`.

`str`: a Java string object, must not be `NULL`.

`start`: the index of the first unicode character in the string to copy. Must be greater than or equal to zero, and less than the string length.

`len`: the number of unicode characters to copy. Must be greater than zero, and "`start + len`" must be less than string length ("`GetStringLength()`").

`buf`: the unicode character buffer into which to copy the string region. Must not be `NULL` if given `len` is > 0.

THROWS:

`StringIndexOutOfBoundsException`: on index overflow.

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.

SEE ALSO:

GetStringChars

ReleaseStringChars

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, must not be NULL.

array: a Java array object, must not be NULL.

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, must not be NULL.

length: array size, must be >= 0.

elementClass: array element class, must not be NULL.

initialElement: initialization value, may be a NULL 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, must not be NULL.

array: a Java array, must not be NULL.

index: array index, must be >= 0 and less than array length ("GetArrayLength()").

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, must not be `NULL`.

`array`: a Java array, must not be `NULL`.

`index`: array index, must be >= 0 and less than array length (`"GetArrayLength()"`).

`value`: the new value, may be a `NULL` 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. The following table describes the specific primitive array constructors. You should replace *New<PrimitiveType>Array* with one of the actual primitive array constructor routine names from this table, and replace ArrayType with the corresponding array type for that routine.

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.

New<PrimitiveType>Array Family of Array Constructors

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

***New<PrimitiveType>Array* Routines**	**Index**
NewFloatArray()	181
NewDoubleArray()	182

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

length: the array length, must be >= 0.

RETURNS:

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

THROWS:

OutOfMemoryError: if the system runs out of memory.

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 following 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.

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.

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, must not be NULL.

array: a Java array object, must not be NULL.

isCopy: a pointer to a boolean, may be a NULL value.

RETURNS:

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

THROWS:

OutOfMemoryError: if the system runs out of memory.

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:

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" as 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. If JNI_COMMIT is passed as the mode argument when elems is a copy of the elements in array, then a final call to *Release<PrimitiveType>ArrayElements* passing a mode argument of "0" or JNI_ABORT, should be made to free the elems buffer.

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 the following table.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

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.

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, must not be NULL.

array: a Java array object, must not be NULL.

elems: a pointer to array elements, as returned by previous *Get<PrimitiveType>ArrayElements* call.

mode: the release mode: 0, JNI_COMMIT or JNI_ABORT.

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 the following table.
- Replace *ArrayType* with the corresponding array type.
- Replace *NativeType* with the corresponding native type for that routine.

Get<PrimitiveType>ArrayRegion Family of Array Accessor Routines

***Get<PrimitiveType>ArrayRegion* Routine**	**Array Type**	**Native Type**

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.

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, must not be NULL.

array: a Java array, must not be NULL.

start: the starting index, must be greater than or equal to zero, and less than the array length (GetArrayLength()).

len: the number of elements to be copied, must be greater than or equal to zero, and "start + len" must be less than array length ("GetArrayLength()").

buf: the destination buffer, must not be NULL.

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 following table.
- Replace *ArrayType* with the corresponding array type.

- Replace *NativeType* with the corresponding native type for that routine.

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.

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, must not be NULL.

array: a Java array, must not be NULL.

start: the starting index, must be greater than or equal to zero, and less than the array length (GetArrayLength()).

len: the number of elements to be copied, must be greater than or equal to zero, and "start + len" must be less than array length ("GetArrayLength()").

buf: the source buffer, must not be NULL.

THROWS:

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

Note: 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. The *Get/Release<primitivetype>ArrayCritical* functions 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.

GetPrimitiveArrayCritical

LINKAGE:

Linkage Index 222 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be `NULL`.

`array`: a Java array, must not be `NULL`.

`isCopy`: a pointer to a boolean, may be a `NULL` value.

RETURNS:

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

THROWS:

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

SINCE:

JDK/JRE 1.2

ReleasePrimitiveArrayCritical

LINKAGE:

Linkage Index 223 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

array: a Java array, must not be NULL.

carray: critical array pointer as returned by GetPrimitiveArrayCritical.

mode: the release mode (see Primitive Array Release Modes): 0, JNI_COMMIT or JNI_ABORT. Ignored if carray was a not copy.

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, ...);`

Be aware that RegisterNatives can change the documented behavior of the JVM (including cryptographic algorithms, correctness, security, type safety), by changing the native code to be executed for a given native Java method. Therefore use applications that have native libraries utilizing the RegisterNatives function with caution.

LINKAGE:

Index 215 in the JNIEnv interface function table.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

clazz: a Java class object, must not be NULL.

methods: the native methods in the class, must not be NULL.

nMethods: the number of native methods in the class, must be greater than zero.

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, must not be NULL.

clazz: a Java class object, must not be NULL.

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, must not be NULL.

obj: a normal Java object or class object, must not be NULL.

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, must not be NULL.

obj: a normal Java object or class object, must not be NULL.

RETURNS:

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

THROWS:

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 buffers in the NIO package and the specification of the java.nio.ByteBuffer class.

Three functions 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. The byte order of the returned buffer is always big-endian (high byte first; java.nio.ByteOrder.BIG_ENDIAN).

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 JNI interface pointer, must not be NULL.

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.

THROWS:

`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 JNI interface pointer, must not be `NULL`.

`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 JNI interface pointer, must not be `NULL`.

`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.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

`method`: a `java.lang.reflect.Method` or `java.lang.reflect.Constructor` object, must not be NULL.

RETURNS:

A JNI method ID that corresponds to the given Java reflection method, or NULL if the operation fails.

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.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

`field`: a `java.lang.reflect.Field` object, must not be NULL.

RETURNS:

A JNI field ID that corresponds to the given Java reflection `field`, or NULL if the operation fails.

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.

PARAMETERS:

env: the JNI interface pointer, must not be NULL.

`cls`: a Java class object, must not be NULL.

`methodID`: a method ID, must not be NULL.

`isStatic`: denotes whether the given `methodID` is a static method.

RETURNS:

Returns an instance of the `java.lang.reflect.Method` or `java.lang.reflect.Constructor` which corresponds to the given `methodID`, or `NULL` if the operation fails.

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.

PARAMETERS:

`env`: the JNI interface pointer, must not be `NULL`.

`cls`: a Java class object, must not be `NULL`.

`fieldID`: a field ID, must not be `NULL`.

`isStatic`: denotes whether the given `fieldID` is a static field.

RETURNS:

Returns an instance of the `java.lang.reflect.Field` which corresponds to the given `fieldID`, or `NULL` if the operation fails.

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, must not be `NULL`.

`vm`: a pointer to where the result should be placed, must not be `NULL`.

RETURNS:

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

Copyright © 1993, 2022, Oracle and/or its affiliates, 500 Oracle Parkway, Redwood Shores, CA 94065 USA.
All rights reserved. Use is subject to license terms and the documentation redistribution policy. Modify Cookie Preferences. Modify Ad Choices.