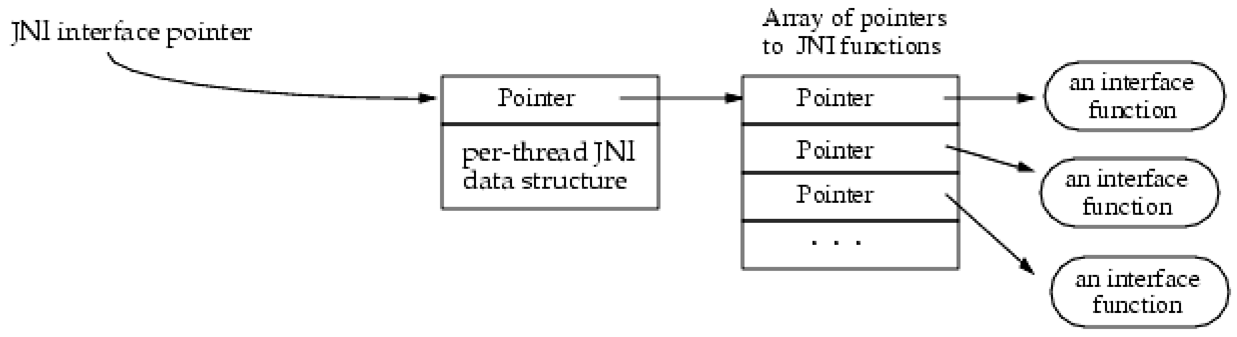
Chapter 2: Design Overview

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

This chapter covers the following topics:

JNI Interface Functions and Pointers

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



Interface pointer

Description of Figure Interface Pointer

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

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

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

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

Compiling, Loading and Linking Native Methods

Since the Java VM is multithreaded, native libraries should also be compiled and linked with multithread aware native compilers. For example, the `-mt` flag should be used for C++ code compiled with the Sun Studio compiler. For code complied with the GNU gcc compiler, the flags `-D_REENTRANT` or `-D_POSIX_C_SOURCE` should be used. For more information please refer to the native compiler documentation.

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

```
package p.q.r;

class A {
    native double f(int i, String s);
    static {
        System.loadLibrary("p_q_r_A");
    }
}
```

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

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

Support for both dynamically and statically linked libraries, and their respective lifecycle management "*load*" and "*unload*" function hooks are detailed in the Invocation API section on *Library and Version Management*.

Resolving Native Method Names

The JNI defines a 1:1 mapping from the name of a `native` method declared in Java to the name of a native method residing in a native library. The VM uses this mapping to dynamically link a Java invocation of a `native` method to the corresponding implementation in the native library.

The mapping produces a native method name by concatenating the following components derived from a `native` method declaration:

1. the prefix Java_
2. given the binary name, in internal form, of the class which declares the `native` method: the result of escaping the name.
3. an underscore ("_")
4. the escaped method name
5. if the `native` method declaration is overloaded: two underscores ("__") followed by the escaped parameter descriptor (JVMS 4.3.3) of the method declaration.

Escaping leaves every alphanumeric ASCII character (`A-Za-z0-9`) unchanged, and replaces each UTF-16 code unit in the table below with the corresponding escape sequence. If the name to be escaped contains a surrogate pair, then the high-surrogate code unit and the low-surrogate code unit are escaped separately. The result of escaping is a string consisting only of the ASCII characters `A-Za-z0-9` and underscore.

UTF-16 code unit	**Escape sequence**
Forward slash (/, U+002F)	_

UTF-16 code unit	Escape sequence
Underscore (_, U+005F)	_1
Semicolon (;, U+003B)	_2
Left square bracket ([, U+005B)	_3
Any UTF-16 code unit `\u`*WXYZ* that does not represent alphanumeric ASCII (`A-Za-z0-9`), forward slash, underscore, semicolon, or left square bracket	`_0wxyz` where `w`, `x`, `y`, and `z` are the lower-case forms of the hexadecimal digits `W`, `X`, `Y`, and `Z`. (For example, `U+ABCD` becomes `_0abcd`.)

Escaping is necessary for two reasons. First, to ensure that class and method names in Java source code, which may include Unicode characters, translate into valid function names in C source code. Second, to ensure that the parameter descriptor of a `native` method, which uses ";" and "[" characters to encode parameter types, can be encoded in a C function name.

When a Java program invokes a `native` method, the VM searches the native library by looking first for the short version of the native method name, that is, the name without the escaped argument signature. If a native method with the short name is not found, then the VM looks for the long version of the native method name, that is, the name including the escaped argument signature.

Looking for the short name first makes it easier to declare implementations in the native library. For example, given this `native` method in Java:

```
package p.q.r;
class A {
    native double f(int i, String s);
}
```

The corresponding C function can be named `Java_p_q_r_A_f`, rather than `Java_p_q_r_A_f__ILjava_lang_String_2`.

Declaring implementations with long names in the native library is only necessary when two or more `native` methods in a class have the same name. For example, given these `native` methods in Java:

```
package p.q.r;
class A {
    native double f(int i, String s);
    native double f(int i, Object s);
}
```

The corresponding C functions must be named `Java_p_q_r_A_f__ILjava_lang_String_2` and `Java_p_q_r_A_f__ILjava_lang_Object_2`, because the `native` methods are overloaded.

Long names in the native library are not necessary if a `native` method in Java is overloaded by non-`native` methods only. In the following example, the `native` method `g` does not have to be linked using the long name because the other method `g` is not `native` and thus does not reside in the native library.

```
package p.q.r;
class B {
    int g(int i);
    native int g(double d);
}
```

Note that escape sequences can safely begin `_0`, `_1`, etc, because class and method names in Java source code never begin with a number. However, that is not the case in class files that were not generated from Java source code. To preserve the 1:1 mapping to a native method name, the VM checks the resulting name as follows. If the process of escaping any precursor string from the `native` method declaration (class or method name, or argument type) causes a "0", "1", "2", or "3" character from the precursor string to appear unchanged in the result *either* immediately after an underscore *or* at the beginning of the escaped string (where it will follow an underscore in the fully assembled name), then the escaping process is said to have "failed". In such cases, no native library search is performed, and the attempt to link the `native` method invocation will throw `UnsatisfiedLinkError`. It would be possible to extend the present simple mapping scheme to cover such cases, but the complexity costs would outweigh any benefit.

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

Native methods can also be explicitly linked using the `RegisterNatives` function. 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.

Native Method Arguments

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

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

The following code example illustrates using a C function to implement the native method `f`. The native method `f` is declared as follows:

```
package p.q.r;

class A {
    native double f(int i, String s);
    // ...
}
```

The C function with the long name `Java_p_q_r_A_f_ILjava_lang_String_2` implements native method `f`:

```
jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (
     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
     jint i,             /* argument #1 */
     jstring s)          /* argument #2 */
{
     /* Obtain a C-copy of the Java string */
     const char *str = (*env)->GetStringUTFChars(env, s, 0);

     /* process the string */
     ...

     /* Now we are done with str */
     (*env)->ReleaseStringUTFChars(env, s, str);

     return ...
}
```

Note that we always manipulate Java objects using the interface pointer *env*. Using C++, you can write a slightly cleaner version of the code, as shown in the following code example:

```
extern "C" /* specify the C calling convention */

jdouble Java_p_q_r_A_f__ILjava_lang_String_2 (

     JNIEnv *env,        /* interface pointer */
     jobject obj,        /* "this" pointer */
     jint i,             /* argument #1 */
     jstring s)          /* argument #2 */

{
     const char *str = env->GetStringUTFChars(s, 0);

     // ...

     env->ReleaseStringUTFChars(s, str);

     // return ...
}
```

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

Referencing Java Objects

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

Global and Local References

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

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

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

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

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

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

Implementing Local References

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

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

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

Accessing Java Objects

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

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

Accessing Primitive Arrays

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

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

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

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

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

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

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

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

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

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

Accessing Fields and Methods

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

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

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

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

A field or method ID does not prevent the VM from unloading the class from which the ID has been derived. After the class is unloaded, the method or field ID becomes invalid and may not be passed to any function taking such an ID. The native code, therefore, must make sure to:

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

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

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

Calling Caller-Sensitive Methods

A small number of Java methods have a special property called caller sensitivity. A *caller-sensitive* method can behave differently depending on the identity of its immediate caller. For example, AccessibleObject::canAccess needs to know the caller to determine accessibility.

When native code calls such a method, there may not be any Java caller on the call stack. It is the responsibility of the programmer to know whether the Java methods being called from their native code are caller-sensitive, and how those methods will respond if there is no Java caller. If necessary the programmer can provide Java code for the native code to call, which in turn calls the original Java method.

Reporting Programming Errors

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

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

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

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

Java Exceptions

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

Exceptions and Error Codes

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

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

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

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

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

Asynchronous Exceptions

One thread may raise an asynchronous exception in another thread by calling the `Thread.stop()` method, which has been deprecated since Java 2 SDK release 1.2. Programmers are strongly discouraged from using `Thread.stop()` as it generally leads to an indeterminate application state.

Furthermore, the JVM may produce exceptions in the current thread without being the direct result of a JNI API call, but because of various JVM internal errors, for example: `VirtualMachineError` like `StackOverflowError` or `OutOfMemoryError`. These are also referred to as asynchronous exceptions.

Asynchronous exceptions do not immediately affect the execution of the native code in the current thread, until:

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

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

Native methods should insert `ExceptionOccurred()` checks in necessary places, such as in any long running code without other exception checks (this may include tight loops). This ensures that the current thread responds to asynchronous exceptions in a reasonable amount of time. However, because of their asynchronous nature, making an exception check before a call is no guarantee that an asynchronous exception won't be raised between the check and the call.

Exception Handling

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

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

After an exception has been raised, the native code must first clear the exception before making other JNI calls. When there is a pending exception, the JNI functions that are safe to call are:

```
ExceptionOccurred()
ExceptionDescribe()
ExceptionClear()
ExceptionCheck()
ReleaseStringChars()
```

```
ReleaseStringUTFChars()
ReleaseStringCritical()
Release<Type>ArrayElements()
ReleasePrimitiveArrayCritical()
DeleteLocalRef()
DeleteGlobalRef()
DeleteWeakGlobalRef()
MonitorExit()
PushLocalFrame()
PopLocalFrame()
DetachCurrentThread()
```