



## Chapter 6

# Interop 1: JNI

This chapter introduces you to the essentials of using the Java Native Interface (JNI) to integrate Java code with platform-specific code. For further coverage of how to use JNI to build a more intuitive, robust communication layer between Java and Win32/COM components, see Appendix A.

### 6.1 Why Interoperate?

A key goal of Java is to provide a consistent platform and API over the specific hardware and software of a particular machine. To the extent that this platform functions correctly and delivers services needed by programmers, Java achieves its objective of being write-once, run-anywhere (WORA). Developers write a single code base, which functions correctly across multiple processors and operating systems, saving the cost of redeveloping the same basic logic over and over again for different platforms. Organizations that adopt Java often value cross-platform code so highly that they insist that *all* new development be done in Java. In this brave new world, there is no need to interoperate with other languages or environments because Java replaces everything else.

Now, back to the real world. While 100 percent pure Java is a laudable objective on some kinds of projects, it is usually unrealistic. No single tool, even Java, is perfect for every task. Single-tool aficionados should heed Mark Twain's warning: "When all you have is a hammer, all problems start to look like nails." Here are several tasks for which Java is ill-suited:

- Direct access to system memory. Java deliberately prevents direct access to system memory, which in most situations is a great benefit in reducing



bugs. However, some hardware devices are queried and controlled through direct access to memory.

- Accessing platform-specific resources. Java provides a standard API for services that most operating systems provide, such as file and network access. However, by definition, Java cannot provide a cross-platform API to a platform-specific service, such as the Windows Registry.
- Accessing code libraries not written in Java. Many organizations already have a large base of well-understood, field-tested code. Even if they wanted to port all this code to Java, the development time and cost would likely be prohibitive.
- Hand-tuned, peak performance code. Java's performance is sufficient for many types of applications being written today. However, there always have been, and probably always will be, certain code paths that need to be hand-tuned in a systems programming language, such as C++, or even in an assembly language.

All of these examples share a common theme, which is the occasional need to escape the confines of the virtual machine. A well-designed hybrid system gets the best of both worlds: 90 percent of the code lives inside a virtual machine, where it enjoys the comforts of runtime type safety, memory protections, and discretionary access control. The remaining 10 percent visits the hostile space outside of the virtual machine, but only long enough to provide some specialized service.

A poorly designed hybrid system, on the other hand, experiences all the problems of both Java and native code: the overhead of the virtual machine plus the mysterious failures endemic to systems programming. Managing the boundary between Java and native code requires careful attention to preserving the benefits of each environment.

JNI allows a Java virtual machine to share a process space with platform native code, typically written in C or C++. From Java, you can find, load, and invoke a native language method, free of the rules of the virtual machine. The converse is also true; from a native language, you can start a virtual machine and then find, load, and invoke methods on Java classes.

JNI itself is a cross-platform standard that is provided by any compliant Java virtual machine. However, the things that you do with JNI are usually platform-specific. For purposes of this chapter, I am going to use C++ and the Microsoft Windows platform for JNI examples. This choice of language and OS probably represents the most common use of JNI, but the examples should be representative of issues you encounter with other languages or platforms as well.

Because C++ and Java are radically different development environments, there are several subtleties to consider when crossing the border between the virtual machine and native code. This chapter will cover the most important of these: the dangers of native code, loading native code, method invocation, error handling, and resource management.

## 6.2 The Dangers of Native Code

The dangers are the easiest part to understand. When you leave the virtual machine, you leave behind all its built-in protections. Java's memory protections do not apply in native code, so native methods can corrupt memory or the VM itself. Java's security checks do not apply in native code, so your native methods operate with essentially "all permissions." Type safety is a fiction in native code, so feel free to treat a `CARM` like a `CLEg` if you like.<sup>1</sup>

Once you internalize the fact that the virtual machine has no control over the behavior of native code, other JNI design decisions make more sense. For example, JNI allows native code to bypass language protection modifiers and to invoke methods nonvirtually. This would seem horribly dangerous if you had not accepted that JNI is *innately* a dangerous environment when compared to pure Java.

Since JNI is so dangerous, why use it at all? In general, you should avoid it. If Java provides a service, there is typically no reason to reimplement that service in native code. Most of the services you will need are already in the Java APIs: files, sockets, databases, user interfaces, security, and so on. But when

---

1. In a type-safe world, your `CHand` would get tired and callused. In native code, such a type gaffe will cause a memory fault—if you are lucky.

you do need JNI, you *really* need it. Simply make sure that you enter a relationship with JNI with your eyes open. JNI will make your Java projects more expensive to develop and maintain. Poorly written native code can make an otherwise excellent Java system totally unreliable. JNI puts the entire native platform under your control; use that power sparingly and wisely.

### 6.3 Finding and Loading Native Code

Bridging between Java and native code is both a logical and a physical problem. The logical problem is one of disparate naming and typing systems. To solve this problem, JNI defines a complete, unambiguous mapping from Java names and types to C++ names and types.<sup>2</sup> The physical problem is finding and loading the appropriate native binary. The process of finding and loading native code is very similar to the process of loading Java classes. Both processes are well defined, but they tend to produce cryptic errors and be poorly understood by developers. This section covers the logical and physical mapping between Java and native code, and it shows how to troubleshoot the most common problems.

JNI is normally used to provide native implementations of methods declared in Java. The `native` keyword indicates that a particular method has a native implementation, as shown in the `getAnswer` method of Listing 6–1.

#### Listing 6–1 The `native` Keyword

```
package com.develop;
public class UltimateQuestion {
    public static native int getAnswer();
}
```

When the virtual machine encounters a call to `getAnswer`, it will expect that the method has already been successfully coded, compiled, linked, located, and loaded. JNI does not provide an implicit mechanism such as the class loader architecture, so you must manually execute each of these steps.

---

2. JNI does *not* define a complete mapping from C/C++ names and types back into Java.



### 6.3.1 Name Mappings

JNI defines a specific name mapping from Java method names to C++ library entry points. For a simple method call, the C++ name should be

```
Java_package_name_classname_methodname
```

So, a C++ implementation of `getNative` would have the name

```
Java_com_develop_UltimateQuestion_getNative
```

Return and parameter types pose a more complex problem. In Java, the size of every type is well defined, but in C++ type sizes can vary from platform to platform. JNI introduces a layer of indirection to deal with this. For every Java primitive type `foo`, JNI declares a C++ type `jfoo` in a platform-specific header file. The platform header uses a `typedef` to map `jfoo` to the matching sized C++ type on that platform. For example, Listing 6–2 shows how JNI handles `int` on a Win32 platform.

#### Listing 6–2 Platform-Specific Native Mapping of Java `int` Type

```
//from jni.h, located in ${JAVA_HOME}/include
#include "jni_md.h"

//jni_md.h is a platform-specific header file
//this is from the version in ${JAVA_HOME}/include/win32
typedef long jint;
```

On all platforms, the JNI representation of `int` will be called `jint`. As you can see here, the Win32 implementation of `jint` is simply a `long`. Other platforms might define `jint` differently.

### 6.3.2 Type Mappings

JNI disposes of language differences in the primitive types with ease. It is when you look at passing object references between Java and C++ that you begin to see how different the languages really are. C++ allows arbitrary pointer indirection, distinguishes between structures and classes,<sup>3</sup> permits

---

3. The distinction between the `struct` and `class` keywords is minimal: Structures default to public access while classes default to private. However, structs and classes tend to be used in different ways, and there is no obvious way to capture this in Java.



multiple inheritance, and supports passing parameters by reference or by value. Java has no pointers, has no notion of a structure as distinct from a class, allows only single implementation inheritance, and always passes parameters by value. These differences cause major problems for any generic mapping between Java and C++ objects:

- Since Java does not permit multiple inheritance, there is no easy way to represent an arbitrary C++ class hierarchy in Java.
- Although Java always passes parameters by value, the value that is copied onto a method stack is a reference to the object. Methods have their own copy of the reference, but they share access to the referenced object. As a result, changes made through the reference are visible outside the method. This is usually desirable for C++ “classes” that encapsulate complex behavior and state, but it may be inappropriate for simple C++ “structs” that are simply typed collections of data.<sup>4</sup>
- If a C++ parameter contains multiple levels of indirection, or if it is used to return information to the caller, it is often unclear what the corresponding Java method declaration should look like.

Many of these difficulties stem from the ambiguities permitted by the C++ type system. Consider the following C++ method declarations:

```
void foo(char** arg);  
void bar(void* arg);
```

Each of these methods has many possible interpretations. The argument to `foo` might be a two-dimensional array of characters, an array of null-terminated strings, or an address that the method will assign to point to a single null-terminated string. The argument to `bar` could be anything at all. From a Java perspective, the problem is a lack of metadata.

There are many possible solutions to the problem of mapping types between different languages such as Java and C++. The most complete solutions

---

4. Remember that the C++ keywords `class` and `struct` do not mandate this distinction in intended usage.

supplement the metadata available at the language level with additional metadata to resolve ambiguities between language-level type systems. CORBA and DCOM are examples of this approach; both use an interface definition language (IDL) to completely describe method types and provide a suite of tools to generate the appropriate language mappings.

For better or worse, JNI takes a much simpler approach. JNI does not define *any mapping at all* from C++ types to Java types. JNI defines a complete but minimalist mapping from Java types to C++ types. Instead of representing Java objects as C++ classes or structures, JNI maps Java objects to opaque handles, which are declared as type `jobject` and subclasses. From C++, you can pass these handles to helper functions implemented by a VM-provided object called the JNI environment pointer. The JNI environment pointer, typed as `JNIEnv*`, is a C++ vtable of callback functions that allow reflective manipulation of Java objects.

The `jobject` and `JNIEnv` types are covered in §6.4. For now, the important detail is that every Java native method translates to a C++ method with two extra arguments: the `JNIEnv*` for calling back into the VM, and the `jobject` `this` reference for the method. For example, the declaration for the native implementation of the `getAnswer` method from Listing 6–1 is

```
JNIEXPORT jint JNICALL
Java_com_develop_UltimateQuestion_getAnswer(JNIEnv*, jclass);
```

The `int` return type in Java has been converted to a C++ `jint`, as expected. The Java method declared no arguments, so the C++ implementation has only the two “extra” arguments mandated by JNI. Note that since the method was declared static, there is no `this` and the second argument is a `jclass` subtype of `jobject`, which references the `UltimateQuestion` class. Additional arguments, if there had been any, would have followed these two. The `JNIEXPORT` and `JNICALL` macros are platform-specific and are used to hide the platform-specific details of correctly exporting an entry point that the VM can link dynamically.



### 6.3.3 Overloaded Names

The naming and type-mapping convention described thus far is fine for most methods, but it breaks down for overloaded methods with the same name. For such methods, JNI defines an additional name-mangling scheme, whereby the method parameters are encoded as valid C++ name characters and tacked onto the end of the method name. For example, consider the following overloaded Java methods:

```
public native void causeConfusion(String arg);  
public native void causeConfusion(int[] arg);
```

For overloaded methods, arguments are encoded in a multistep process:

1. Append a double underbar (\_\_) to the nonoverloaded version of the type name.
2. Select the type name used internally by the VM, as shown in Table 6–1.
3. Escape any characters that would be illegal in C++ names, using the escape codes shown in Table 6–2.

**Table 6–1 Virtual Machine Type Names**

Java Type	VM Name
int	I
float	F
long	J
double	D
byte	B
boolean	Z
short	S
char	C
<i>anytype</i> []	[ <i>anytype</i>
<i>somepkg.SomeClass</i>	L <i>somepkg.SomeClass</i> ;





**Table 6–2 JNI Name Mangling of non-C++ Type Names**

Character	Mangled Form
_	_1
;	_2
[	_3
Unicode with hex value XXXX	_OXXXX

The correct JNI method declarations for the `causeConfusion` methods look like this when wrapped to fit the printed page:

```
JNIEXPORT void JNICALL
Java_com_develop_UltimateQuestion_causeConfusion__Ljava_lang_Stri
ng_2(JNIEnv *, jobject, jstring);

JNIEXPORT void JNICALL
Java_com_develop_UltimateQuestion_causeConfusion___3I(
JNIEnv *, jobject, jintArray);
```

This gets ugly fast—they don’t call it name mangling for nothing. Fortunately, you do not need to generate these method declarations by hand. The Java SDK includes a command-line tool, `javah`, that extracts the metadata for the native methods in a list of classes and generates an appropriate header file. For example, the `javah` command

```
javah -d ../cpp com.develop.UltimateQuestion
```

uses the metadata in the `UltimateQuestion` class to generate the header file `../cpp/UltimateQuestion.h`. To avoid errors when you are linking to native code, you should always use `javah` to generate the correct method names.

### 6.3.4 Loading Native Libraries

The naming conventions, plus the mappings for primitive and object types, solve the logical problem. The remaining problem is a physical one; that is, where does the virtual machine look to find implementations of native methods? The first important point is that there is no association between the native library

name and the methods it contains. It is reasonable and common to implement `Foo`'s native methods in a library named `Foo`. However, it is equally reasonable to group all native methods for an entire application in a library named `App`. It is perverse, but still legal, to place the implementation of `Foo`'s native methods in a library named `Bar`, or to spread them across several libraries. No matter, the virtual machine will automatically match C++ entry points to Java methods using the naming conventions described above. The only thing you have to do is call a JNI load API that loads the requisite native library before a particular native method is actually invoked.

The lowest-level load API is `Runtime.load`, which takes a full path to the shared library. Assuming that you have compiled a C++ implementation of the `UltimateQuestion` methods into a `d:\shared` directory on a Win32 machine, the Java code in Listing 6–3 would cause the library to be loaded:

### Listing 6–3 Linking with Full Paths

```
//EXAMPLE ONLY. DO NOT USE HARD-CODED PATH NAMES!
public class HardPathClient {
    public static void main(String [] args) {
        UltimateQuestion uq = new UltimateQuestion();
        Runtime.getRuntime().load(
            "d:/shared/UltimateQuestion.dll");
        System.out.println(uq.causeConfusion("babble"));
    }
}
```

Notice that the call to `load` occurs after an `UltimateQuestion` instance has already been created. Native methods can be loaded at any time: before loading the associated Java class, while loading the class, or after creating instances of a class. Native methods can even be loaded after a prior attempt to invoke them throws an error.

Despite its simplicity, the code above is unmaintainable and should be avoided. The placement of a hard-coded path in the Java source code ties method loading to a compile-time decision, which is contrary to the spirit of Java. Avoid `Runtime.load`.

The preferred approach to loading native code is the `System.loadLibrary` method. Although `loadLibrary` takes a string argument, just

as `Runtime.load` does, the argument is interpreted very differently. The string passed to `loadLibrary` is a library name, not a full path. In fact, path names and extensions are illegal in strings passed to `loadLibrary`. With `loadLibrary`, you specify only the short name of the library, and the path to search is controlled through virtual machine options. In order to make library dependencies more clear, and to guarantee that native libraries are loaded in time to service native method calls, you typically place the call to `loadLibrary` in a static initializer for the class that declares the native methods, as shown here:

```
public class UltimateQuestion {  
    static { System.loadLibrary("UltimateQuestion"); }  
    //as before...  
}
```

This style has several advantages over the approach that used `Runtime.load`. Because the native library is loaded in a static initializer of the declaring class, clients of `UltimateQuestion` do not have to worry about loading the native methods. The use of the short form name of the library allows the virtual machine to apply an appropriate name translation for the current platform, for example, `UltimateQuestion.dll` for Win32, `libUltimateQuestion.so` for UNIX, and so on. Most importantly, the short form name leaves the exact load path as a runtime configuration issue.

The native library load path is specified by the `java.library.path` system property. You can specify this property yourself on the command line for the Java launcher, like this:

```
java -Djava.library.path=../UltimateQuestion/ \  
UltimateQuestionClient
```

If you do not specify a library path, the virtual machine will establish a search path in a platform-dependent fashion. Currently, on Solaris the Sun virtual machine uses the value of the environment variable `LD_LIBRARY_PATH`, and on Win32 it uses the value of `PATH`. Do not rely on these environment settings in a production system. Environment variables are subject to deliberate or accidental modification by users and other programs; so far, the history of Java has seen a constant shift away from environment variables and toward explicit arguments to

the command-line tools. In Java, environment variables are a useful crutch for beginners, but nothing more. Minimize the potential for confusion by always specifying the load path on the command line or from a shell script.

### 6.3.5 Class Loaders and JNI

Native code has a complex relationship with the class loader delegation model. `System.loadLibrary` delegates to the current class loader to actually load the library; the code that analyzes `java.library.path` actually lives in the `ClassLoader` class. Class loaders add three wrinkles to the basic native loading story: the `findLibrary` method, the `sun.boot.library.path` option, and the handling of multiple libraries.

First, a class loader can override the `findLibrary` method, shown in Listing 6–4, to augment the normal library search algorithm.

#### Listing 6–4 The find Library Method

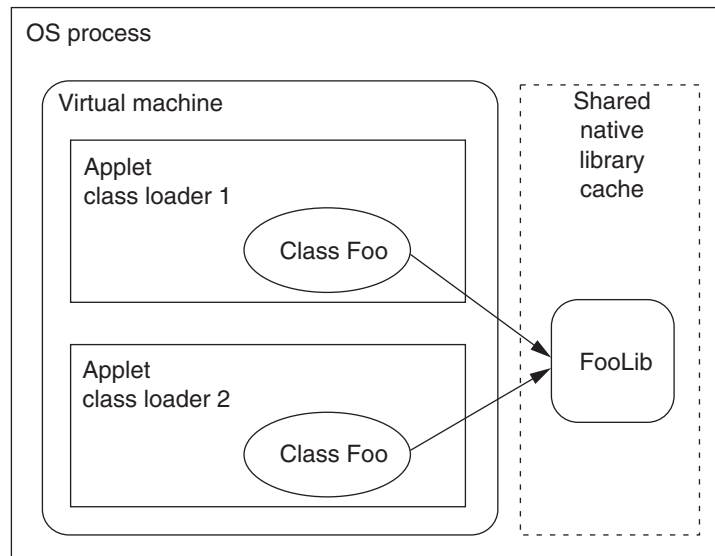
```
package java.lang;
public class ClassLoader {
    protected String findLibrary(String libname) {
        return null;
    }
    //remainder omitted for clarity
}
```

If a class loader implementation overrides this method and returns a full path, then that path is passed to `Runtime.load` to load the library.

Second, `ClassLoader` checks another path before consulting `java.library.path`. Class loaders first check `findLibrary`, then the paths listed on `sun.boot.library.path`, and only after these checks fail does it resort to checking `java.library.path`. Neither `findLibrary` nor the boot path are widely used. Most class loaders do not override `findLibrary`, and `sun.boot.library.path` is intended not for loading application libraries, but for customizing native code used by the virtual machine itself. Use `sun.boot.library.path` to specify alternate locations for VM code such as JIT compilers, and use `java.library.path` to locate application native code.

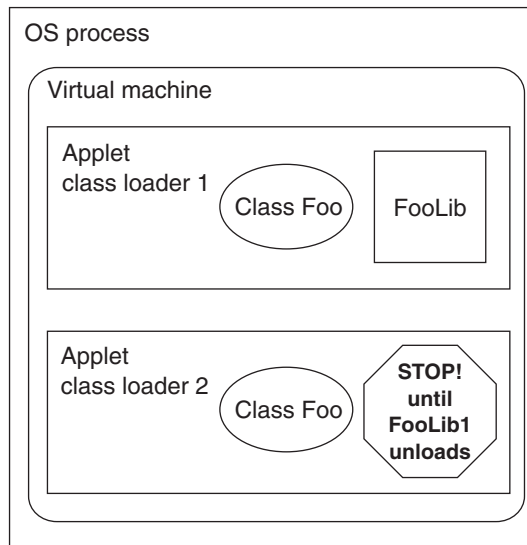
The third, and most important, wrinkle introduced by class loaders is the treatment of multiple versions of native code. Remember that class loader delegations define namespaces (§2.4.4). Classes loaded by a more senior class loader are visible to all subordinate class loaders, allowing code to be shared. Classes loaded by junior class loaders are not visible to each other, thus allowing multiple implementations of the same class to coexist.

Unfortunately, JNI makes a brutal simplification of these delegation rules. In early versions of Java, native libraries ignore the delegation model and are visible across all class loaders. This causes problems since two different versions of a class will get the same native implementations whether they want to or not, as shown in Figure 6–1.



**Figure 6–1** Prior to Java 2, multiple class loaders shared the native cache.

Versions of the Java SDK from 1.2 onward partially fix this problem by making native code visible only to a class loader delegation. This prevents unrelated classes from accidentally or maliciously linking with the wrong native methods. However, the specification now says that the same JNI library cannot be loaded into more than one class loader at any given time (see Figure 6–2). This makes it difficult to dynamically update classes with native methods. If the class loads its



**Figure 6–2 Post Java 2, each delegation has its own native libraries.**

own native methods, then no other class loader can load the same library, at least not until the first class is garbage collected.

There are three workarounds for JNI's weak handling of multiple native library versions:

1. Load native code via a senior class loader (such as the application loader) that is not involved in hot deployment. The downside is that while the classes can be updated on-the-fly, the native methods can never change.
2. Always make sure that an old version of a native library gets unloaded. This requires careful discipline so that a class loader can be garbage collected, plus a virtual machine with an aggressive, reliable implementation of `System.gc` and `System.runFinalization` functionality.
3. Build your own scheme that avoids collisions by incrementing the library name each time a native library is redeployed.

All of these are gross hacks when compared to the elegance of Java class loading. The unfortunate truth is that interoperability is a fairly low priority, and JNI has evolved just enough to (partially) deflect criticism from developers.

### 6.3.6 Common Errors Loading Native Libraries

There are several errors associated with loading and linking to native code. The most basic problem is failing to call `loadLibrary` before executing a native method, which leads to a fairly obvious `UnsatisfiedLinkError` at the point where you attempt to invoke the method. In Listing 6–5, the `ForgetToLoadLibrary` class demonstrates this problem:

#### Listing 6–5 Forgetting to Call `loadLibrary`

```
public class ForgetToLoadLibrary {
    public static native void neverLoaded();
    public static void main(String [] args) {
        neverLoaded();
    }
}
>java ForgetToLoadLibrary
>java.lang.UnsatisfiedLinkError: neverLoaded
  at ForgetToLoadLibrary.neverLoaded(Native Method)
```

If `loadLibrary` fails to find a library file matching the name specified, a different error is thrown at the point of the `loadLibrary` call, as shown by the `LoadNonExistentLibrary` example in Listing 6–6:

#### Listing 6–6 Loading a Non Existent Library

```
public class LoadNonExistentLibrary {
    public static native void neverLoaded();
    public static void main(String [] args) {
        System.loadLibrary("DoesNotExist");
        neverLoaded();
    }
}
>java LoadNonExistentLibrary
>java.lang.UnsatisfiedLinkError: no DoesNotExist in
java.library.path
  at java.lang.ClassLoader.loadLibrary(ClassLoader.java:1312)
```

Both of the problems generate an `UnsatisfiedLinkError`, which is an unchecked subclass of `Error`. The javadoc documentation for `Error` states that an `Error` “indicates serious problems that a reasonable application should not try to catch.” `UnsatisfiedLinkError` is an obvious exception to this

rule. It is entirely reasonable to write an application that checks to see if some native-based service is available and then continues down a different path if the service cannot be loaded.

A subtler problem occurs if you make one or more successful calls to `loadLibrary`, but none of them loads an entry point that matches the signature of the native method you are calling. This causes the same `UnsatisfiedLinkError` that you saw in Listing 6–5, as shown by the `LoadTheWrongLibrary` example in Listing 6–7:

#### Listing 6–7 Loading the Wrong Library

```
public class LoadTheWrongLibrary {
    public static native void neverLoaded();
    public static void main(String [] args) {
        //load some unrelated library
        System.loadLibrary("UltimateQuestion");
        neverLoaded();
    }
}
>java -Djava.library.path=UltimateQPath LoadTheWrongLibrary
>java.lang.UnsatisfiedLinkError: neverLoaded
    at LoadTheWrongLibrary.neverLoaded(Native Method)
```

Old symptom, but new problem. The error does not occur until the native method invocation; this indicates that `loadLibrary` found a library but not one that contained the `neverLoaded` method. There are three likely ways that this problem can occur, all involving abuse of the `javah` tool:

1. If you misspell method or type names in the native method declaration, the JNI naming algorithm will not be able to locate the method. Avoid this problem by obtaining the declaration from the `.h` file that `javah` generates.
2. If you are using a C++ compiler, the compiler may do its own name mangling, changing the names to a form that is unrecognizable by the VM. Wrapping your method declarations or implementations in an `extern "C"` block prevents this, and `#include`ing the `javah`-generated `.h` file does this for you automatically.
3. There is a bug in `javah`'s handling of packages. The native declaration for a method should include a prefix based on the package name, as shown in



Listing 6–8. However, `javah` infers the package structure from the current directory instead of from the package name. Inexperienced Java programmers tend to navigate all the way to a class's directory before running `javah`. Instead of generating an error, `javah` produces a corrupt `.h` file that does not include the package names, as shown in Listing 6–9 below. Listing 6–8 shows the correct usage.

#### Listing 6–8 Correct Handling of Package Names

```
package com.develop;
public class PackageDweller {
    public native void nativeMethod();
}

>cd classes
>javah com.develop.PackageDweller

//excerpt from com_develop_PackageDweller.h
//with correct package names
JNIEXPORT void JNICALL
Java_com_develop_PackageDweller_nativeMethod
(JNIEnv *, jobject);
```

#### Listing 6–9 Incorrect Handling of Package Names

```
>cd classes/com/develop
>javah PackageDweller

//excerpt from PackageDweller.h.
//Note package names missing from method
JNIEXPORT void JNICALL Java_PackageDweller_nativeMethod
(JNIEnv *, jobject);
```

### 6.3.7 Troubleshooting Native Loading

There are some tools that can help to diagnose JNI-related bugs. Just as with class loading, Java provides a debugging flag specific for JNI. The `verbose:jni` flag tells the runtime to generate (among other things) console output for every native method loaded. Unfortunately, it does not tell where the method was loaded from or what files were attempted when a native load fails. It would be straightforward to write a custom version of `java.lang.ClassLoader` that

produces a more complete log of native activity and then install the custom version by setting the bootclasspath as described in Chapter 2. Finally, most operating systems have debugging tools that allow you to monitor both file access and loading of shared libraries.

Resorting to any of these techniques is probably overkill because native loading is not nearly as complex as class loading. Now that you are armed with the short list of problem cases above, you should be able to troubleshoot most JNI loading problems by inspection.

The process of loading native code is the source of many beginner headaches, but it is not terribly complex once you know the basics. The naming and search process are important, but arbitrary, details. The one place where the loading process intersects with more significant issues is in the type mappings between Java objects and `jobject` handles. This design decision greatly curtails how JNI can be used, and it is covered more thoroughly in §6.4.

## 6.4 Calling Java from C++

Once you have crossed the boundary into a native method implementation, you can stay there and do whatever the underlying platform will allow. However, to do anything significant, you will probably need to call back into the virtual machine. Any Java object that is passed into native code appears as an opaque `jobject` handle, not as a C++ structure or `vtable`. Therefore, the only way to access fields or methods on a `jobject` is to call back into the virtual machine through the provided `JNIEnv` pointer. Also, because the `jobject` type is opaque at compile time, there is no direct invocation of Java methods or direct access of Java fields. Instead, the `JNIEnv*` provides a set of functions similar to the `Field`, `Method`, and `Constructor` objects in Java reflection.

All JNI access to the virtual machine is reflective. This makes calling back into the virtual machine both tedious to code and slow to execute. The essence of good JNI design is to understand the expense of the boundary crossings between native code and the virtual machine and to minimize them.

As a simple example of the problems at the Java/native boundary, consider the `NativePoint` class as shown in Listing 6–10.

**Listing 6-10 The NativePoint Class**

```

public class NativePoint {
    private int x;
    private int y;
    public String toString() {
        return "x= " + x + " y= " + y;
    }
    public void move(int xinc, int yinc) {
        x += xinc;
        y += yinc;
    }
    public native void nativeMove(int xinc, int yinc);
}

```

In order to implement the `nativeMove` method, you must access the `NativePoint` class, use reflection to discover fields `x` and `y`, then use reflective access to extract the old value of each field, and then use reflective access to set the new value of each field. In total, this requires seven trips from native code back into the virtual machine, as shown here in Listing 6-11.

**Listing 6-11 A Simple Native Method Implementation**

```

//error handling omitted for brevity
JNIEXPORT void JNICALL Java_NativePoint_nativeMove
    (JNIEnv *pEnv, jobject obj, jint xinc, jint yinc)
{
    jclass cls = pEnv->GetObjectClass(obj);
    jfieldID fldX = pEnv->GetFieldID(cls, "x", "I");
    jfieldID fldY = pEnv->GetFieldID(cls, "y", "I");
    int x = pEnv->GetIntField(obj, fldX);
    int y = pEnv->GetIntField(obj, fldY);
    pEnv->SetIntField(obj, fldX, x + xinc);
    pEnv->SetIntField(obj, fldY, y + yinc);
}

```

If you allow for the omnipresent `JNIEnv*` and for stylistic differences between C++ and Java, this code looks very similar to reflective access code written in Java. This example shows manipulation of integer fields; all the other primitive types have similar methods as summarized in Listing 6-12.

For field access, there are only three substantive differences between JNI and Java reflection:

1. The `GetFieldID` method takes an extra parameter that specifies the type of field. The type must be specified using the virtual machine's internal naming scheme, as shown in Table 6–1 earlier in this chapter.
2. All the handle types in the `jobject` family have special lifetime constraints dictated by the needs of the garbage collector. Unless otherwise noted, JNI references to Java objects are method local and thread local. The reasons for this are discussed in more detail under §6.6.
3. JNI does not make the distinction between public and nonpublic fields implied by reflection's `getField` and `getDeclaredField` methods. JNI-style reflection ignores language-level protections at all times.

#### Listing 6–12 JNI APIs for Field Access

```
struct JNIEnv {
//introspection:
jfieldID GetFieldID(jclass clazz, const char *name,
                    const char *sig);
jfieldID GetStaticFieldID(jclass clazz, const char *name,
                           const char *sig);

//access: in the following declarations, replace type with
//object, int, float, long, double, boolean, short, byte, char
jtype GetTypeField(jobject obj, jfieldID fieldID);
void SetTypeField(jobject obj, jfieldID fieldID, jtype val);
jtype GetStaticTypeField(jclass clazz, jfieldID fieldID);
void
SetStaticTypeField(jclass clazz, jfieldID fieldID, jtype val);
//remainder omitted for clarity
};
```

The C++ implementation of `nativeMove` is much more tedious and error-prone to write than the Java implementation of `move`, which does essentially the same task. However, C++ programmers are expert in using macros to hide such tedium, so in the long run, this might not be a significant issue.

More important is the performance penalty for making so many crossings back into the virtual machine. In a simple test, the `move` method took approximately 40 nsec, while the `nativeMove` method took 5,000 nsec, which is over

100 times slower. For comparison, you can run the same test on your own configuration using the `TimePoint` and `TimeNativePoint` classes included on the website for this book [Hal01]. The relative numbers can vary widely depending on your virtual machine and hardware, but the native method should always be slower.

There are two conclusions to be drawn from this result. First, you should *not* use JNI to try to speed up small-grained methods. Even if your C++ implementation of an algorithm runs faster than the same algorithm in Java, the overhead of crossing the JNI boundary will overwhelm any language difference for small methods. Second, you should write your code to minimize the number of round trips from Java to native code.

#### 6.4.1 Minimizing Round Trips

Look back at the implementation of `nativeMove` from Listing 6–11, and you will see that the last four method calls are unavoidable. However, the `jfieldID` values are valid for the lifetime of a class, so these values can and should be precalculated. As of SDK 1.2, JNI provides the perfect hook for this, via the `JNI_OnLoad` method. Listing 6–13 demonstrates an improved `nativeMove` implementation that precalculates the `jfieldIDs` when the library is loaded.

#### Listing 6–13 Improving JNI Performance by Precalculating `jfieldIDs`

```
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* pEnv;
    if (JNI_OK != vm->GetEnv((void **)&pEnv, JNI_VERSION_1_2))
    {
        return JNI_EVERSION;
    }
    jclass cls = pEnv->FindClass("NativePoint");
    s_fldX = pEnv->GetFieldID(cls, "x", "I");
    s_fldY = pEnv->GetFieldID(cls, "y", "I");
    return JNI_VERSION_1_2;
}

JNIEXPORT void JNICALL Java_NativePoint_nativeMove
(JNIEnv* pEnv, jobject obj, jint xinc, jint yinc)
{ //use the fieldIDs precalculated in JNI_OnLoad
    int x = pEnv->GetIntField(obj, s_fldX);
    int y = pEnv->GetIntField(obj, s_fldY);
```

```

    pEnv->SetIntField(obj, s_fldX, x + xinc);
    pEnv->SetIntField(obj, s_fldY, y + yinc);
}

```

The documented purpose of `JNI_OnLoad` is to return the version of JNI that the library expects, in this case `JNI_VERSION_1_2`. The method also provides a perfect place to cache values that will be useful for the lifetime of the library, such as `jfieldIDs`. On my test machine, this new version of `nativeMove` is five times faster than the original (but still twenty times slower than the all-Java `move` method).

Given this performance disparity, you will probably only use JNI for tasks that are simply impossible to perform in Java. For example, imagine that the `NativePoint` class actually controls the location of a robot on a grid. The custom software for controlling the object is only exposed as a C-style API, so you cannot move the robot directly from Java. Listing 6-14 shows a version of `nativeMove` that could be used to move the robot.

#### Listing 6-14 Using JNI to Call a C-Style API

```

static jmethodID s_methMove;
//stubbed-out robot API
void moveRobot(int xinc, int yinc) {
}
JNIEXPORT void JNICALL Java_NativePoint_nativeMove
    (JNIEnv* pEnv, jobject obj, jint xinc, jint yinc)
{
    moveRobot(xinc, yinc);
    pEnv->CallVoidMethod(obj, s_methMove, xinc, yinc);
}
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved) {
    JNIEnv* pEnv;
    if (JNI_OK != vm->GetEnv((void **)&pEnv, JNI_VERSION_1_2)) {
        return JNI_EVERSION;
    }
    jclass cls = pEnv->FindClass("NativePoint");
    s_methMove = pEnv->GetMethodID(cls, "move", "(II)V");
    return JNI_VERSION_1_2;
}

```

There are several new things transpiring here. Notice that instead of calling back into the virtual machine four times to get and set `x` and `y`, the `native-Point` implementation is simply leveraging the Java implementation of `move`. Just as with field access, JNI method access looks very similar to reflection. The `GetMethodID` method returns a `jmethodID`, which should be cached just as `jfieldIDs` are. The `CallVoidMethod` callback is used to invoke the method, and it is one of a large family of similarly named methods that vary by the return type and by the C++ notation for passing an unknown number of arguments (see Listing 6–15).

#### Listing 6–15 JNI APIs for Method Access

```
struct JNIEnv {
//introspection:
jmethodID GetMethodID
    (jclass clazz, const char *name, const char *sig);
jmethodID GetStaticMethodID
    (jclass clazz, const char *name, const char *sig);

//access: Replace type with one of object, int, short, long,
//float, double, boolean, char, int, or void
//(exception to the rule: there is no jvoid, just void)
//three types of each method, for different C/C++ styles
//of passing multiple arguments.
jtype CallTypeMethod(jobject obj, jmethodID methodID, ...);
jtype CallTypeMethodV
    (jobject obj, jmethodID methodID, va_list args);
jtype CallTypeMethodA
    (jobject obj, jmethodID methodID, jvalue * args);
jtype CallStaticTypeMethod(jclass c, jmethodID mid, ...);
jtype CallStaticTypeMethodV
    (jclass c, jmethodID mid, va_list args);
jtype CallStaticTypeMethodA
    (jclass c, jmethodID mid, jvalue * args);
jtype CallNonvirtualTypeMethod(jclass c, jmethodID mid, ...);
jtype CallNonvirtualTypeMethodV
    (jclass c, jmethodID mid, va_list args);
jtype CallNonvirtualTypeMethodA
    (jclass c, jmethodID mid, jvalue *args);
```



### 6.4.2 Performance Comparisons

Table 6–3 summarizes the performance of the various examples shown previously in this chapter. The tests were run on a 1.3 HotSpot VM. Beware that different VMs will have radically different characteristics. If you are using JNI in a performance-sensitive application, you will need to profile your code separately for each target VM.

**Table 6–3 Performance of NativePoint Implementation Strategies**

Implementation	Time (nsec)
Java move implementation	40
nativeMove, no caching	5000
nativeMove, cached jfieldIDs	1000
nativeMove, cached jmethodIDs	7000

These numbers contain a surprise: Using the `move` method turns out to be slower (7000 nsec) than crossing the JNI boundary four times and manipulating the fields directly (1000 nsec). So, in this particular test case, calling a method back in the virtual machine was more expensive than accomplishing the same thing by four reflective field accesses. You might still choose the callback method since it is cleaner and requires less native code.

In many cases the performance variations may not matter at all. For example, if it takes a millisecond to move the robot, then moving the robot will dwarf all the JNI implementation differences. Even if you *do* care about performance differences in the microsecond range, you should take these results with a large cube of salt. The performance of various JNI services will vary dramatically across virtual machines.

### 6.4.3 Differences between JNI and Reflective Invocation

Although JNI method invocation and reflective method invocation are similar, there are a few differences. First, JNI invocation never needs special permission to access private or protected class members; there is no JNI equivalent to





reflection's `AccessibleObject` class. JNI code has complete access to all class members, regardless of Java access modifiers.

JNI's lax rules for reflection are reasonable when you remember that the virtual machine has no real control over native code anyway. If you introduce native code into your process space, that code can do anything. Once allowed inside the native boundary, a determined hacker can bypass any defense the virtual machine might mount, so respecting access modifiers within JNI would provide only a false sense of security.

A second difference between JNI invocation and reflective invocation is that JNI does not have to respect virtual methods. In Java code, a method is either virtual or it is not. Private, static, and some final methods are not virtual; all other methods are virtual.<sup>5</sup> This is true both of direct invocation and reflection. There is no way in the Java language to bypass these rules, although you can chain back up to the immediate base class implementation with the `super` keyword. In JNI, you can choose to ignore the virtualness of a method. For every normal JNI invocation API, there is a corresponding nonvirtual invocation call that resolves to the exact class used to get the `jmethodID`, as shown in Listing 6–16.

#### Listing 6–16 Virtual and Nonvirtual Invocation

```
jclass cls = pEnv->FindClass("NativePoint");
jmethodID mid = pEnv->GetMethodID(cls, "move", "(II)V");
//normal virtual call
pEnv->CallVoidMethod(obj, s_methMove, xinc, yinc);
//this will always call NativePoint's method, even on a
//subclass instance that overrides the method
pEnv->CallNonvirtualVoidMethod(obj, s_methMove, xinc, yinc);
```

It is difficult to imagine a use of this feature that is not a hack, in the disparaging sense of the term. Bypassing Java's notion of virtual methods could violate the expectations of a derived class, leading to arcane bugs. Because of this, you should avoid the `Nonvirtual` forms wherever possible.

5. Of course, a clever virtual machine implementation such as HotSpot can treat a virtual method as nonvirtual if there is only one implementation of the method currently visible. However, this is a performance optimization that would have to be undone if another class's implementation of that virtual method ever did get loaded.

Most JNI code manipulates the fields and methods of existing instances. JNI also provides functions to manipulate class loaders and create new instances, similar to the services of the `java.lang.ClassLoader` and `java.lang.reflect.Constructor` classes. The most important class loader and construction functions are summarized in Listing 6–17.

#### Listing 6–17 JNI Class Loader and Construction Functions

```
struct JNIEnv {
    jclass DefineClass(const char *name, jobject loader,
                      const jbyte *buf, jsize len);
    jclass FindClass(const char *name);
    //plug in primitive type names to generate array API decls:
    jtypeArray NewTypeArray(jsize len);
    jobject NewObject(jclass clazz, jmethodID methodID, ...);
    jobject AllocObject(jclass clazz);
    //remainder omitted for clarity
}
```

`DefineClass` is equivalent to `ClassLoader.defineClass`. `FindClass` is similar to `Class.forName`, and it will find any class visible to the system class loader. The array constructor methods are straightforward, taking the form `NewTypeArray`, where *Type* is replaced by the name of a primitive or by object. `NewObject` is similar to `Constructor.newInstance`.

Notice that `NewObject` borrows the `jmethodID` from JNI method invocation; there is no distinct `jconstructorID` type. In order to find the correct `jmethodID` for a constructor, use the JNI method APIs plus the virtual machine's internal name for a constructor, which is `<init>`, and a phony return type of `void`. For example, to call a constructor for `NativePoint` that takes two `int` arguments you would use the syntax shown in Listing 6–18.

#### Listing 6–18 Calling a Java Constructor from Native Code

```
jmethodID cons = pEnv->GetMethodID(clsNativePoint,
                                     "<init>", "(II)V");
pEnv->NewObject(clsNativePoint, cons, 10, 10);
```

The only JNI object construction method with no counterpart in the Java world is `AllocObject`. `AllocObject` creates a Java instance without invoking any

constructor. This is a very dangerous trick, and it is impossible to do in pure Java code. Most objects rely on constructors to reach an initial valid state. By bypassing the constructor, you risk causing malformed objects that will cause bizarre bugs later on. However, this trick has its uses. If you are re-creating an object from a serialization stream or from some other persistence format, it can be more efficient to skip constructors entirely. The object's state is known to be safe because the object was in a valid state when it was serialized.<sup>6</sup> The Java serialization architecture uses constructorless instantiation to avoid the onerous requirement that all serializable objects have a default constructor. See §4.2.2 for details.

## 6.5 Error Handling in JNI

When two programming platforms meet, you have to deal with all the idiosyncrasies of both. In JNI, this is most obvious when you are dealing with errors and failures. There are at least four distinct issues to consider:

1. What happens to the virtual machine when native code fails?
2. How should JNI code deal with C++ exceptions?
3. How should JNI code deal with Java exceptions?
4. How should JNI code communicate errors back to the VM?

The answer to each of these questions stems from a single principle: Well-written JNI code should preserve the appearance of Java, even when native code fails. In other words, problems should only reach the virtual machine in the form of Java exceptions.

### 6.5.1 Failures in Native Code

The first issue, failures in native code, is important because C and C++ introduce many risks not present in Java code. Most of these risks are caused by using pointers incorrectly. If native code inadvertently addresses the wrong memory locations, there are several possible outcomes. The process may fault and immediately be destroyed by the operating system or the hardware. Or,

---

6. Of course, this assumes that the stream format was not accidentally or maliciously corrupted. See Chapter 4 for details on dealing with this problem.

data may be silently corrupted, causing the virtual machine or operating system to fail mysteriously later. Worse, data may be silently corrupted while everything appears to be normal. JNI cannot protect you from any of these problems; all you can do is write and test native code with extra caution.

One surprising aspect of this danger is that the virtual machine itself is exposed to you as native code, through the `JavaVM` and `JNIEnv` pointers. Normally you access the virtual machine's services through Java, with well-defined guarantees that code will either succeed or throw a well-known exception. Not so with the `JNIEnv` and `JavaVM` pointers. With these, if you pass incorrect arguments to any JNI functions, the results are *not defined* and are likely to be catastrophic. For example, an invalid `jobject` handle might cause the following output from the HotSpot VM:

```
# HotSpot Virtual Machine Error, EXCEPTION_ACCESS_VIOLATION
# Please report this error at
# http://java.sun.com/cgi-bin/bugreport.cgi
#
# Error ID: 4F533F57494E13120E43505002D4
```

If you are doing JNI work, do not rush to report messages like this one as bugs against HotSpot—the bugs are almost certainly yours.

### 6.5.2 Handling C++ Exceptions

The second issue is how to handle C++ exceptions in JNI code. From a programmer's perspective, it would be nice if C++ exceptions were automatically converted into Java exceptions. However, the JNI architecture makes no attempt to derive Java representations for arbitrary C++ objects, so there is no obvious mapping from a C++ exception to a Java exception. Even if there were, there is another problem. While Java is a language and a binary standard, C++ is only a language standard. This means that different C++ compilers can (and do) implement exceptions in slightly different ways.

The lack of a binary standard for C++ exceptions makes it impossible for the Java language to have a one-size-fits-all C++ exception catcher. If Java wanted to catch all C++ exceptions at the native boundary, then JNI would have to include exception-handling code that was rebuilt for each compiler. Rather

than face this complexity, JNI simply disallows throwing C++ exceptions across the C++/Java boundary. Disregard this warning at your peril; the behavior of a C++ exception inside the virtual machine is undefined. If a JNI method might encounter a C++ exception, you should catch that exception in native code to prevent it from destroying the virtual machine. In practice, this means you need a top-level catch block for every JNI method.

### 6.5.3 Handling Java Exceptions from Native Code

The third issue is dealing with Java exceptions that occur while you are in native code. Unless documented otherwise, any `JNIEnv` function can trigger a Java exception. Of course, you will not see the exception directly because there is no mapping from Java exceptions into C++ exceptions. There are two ways to detect that an exception has occurred. With some `JNIEnv` functions, you can infer an error from the return value. For example, `FindClass` will return zero if the class cannot be found. Other methods, such as `CallVoidMethod`, do not have a return value that can be used to indicate an exception. For these, you must call `ExceptionOccurred` or `ExceptionCheck` to detect an exception, as shown in Listing 6–19.

#### Listing 6–19 Detecting a Pending Exception in JNI

```
pEnv->CallVoidMethod(obj, s_methMove, xinc, yinc);

//option 1. get the jobject that represents the exception
jthrowable exc;
if (NULL != (exc = pEnv->ExceptionOccurred())) {
    //run about, scream, and shout...
}

//option 2. peek to see if the exception is pending
if (JNI_TRUE == pEnv->ExceptionCheck()) {
    //more running about...
}
```

The `ExceptionOccurred` call returns a `jthrowable` if an exception is pending, or zero if it is not. Because `jthrowable` is a subtype of `jobject`, you can manipulate it from JNI just as you would any other Java object; for example, you can reflectively discover and use its fields and methods.

When an exception occurs in JNI you have two choices: Either handle the exception from native code, or clean up and get out. If you choose to handle the exception, it is just as if you used a catch block in Java code. The exception is vanquished and execution on the thread can continue. Since JNI provides no C++ mapping of a Java catch block, you must handle exceptions using another API call, `ExceptionClear`:

```
if (0 != (exc = pEnv->ExceptionOccurred())) {
    pEnv->ExceptionClear();
}
```

If you do not handle an exception with `ExceptionClear`, you cannot continue to use the virtual machine from that thread. You must free any resources you need to free, and then exit the native method.

If you do not intend to handle an exception anyway, there is no need to get a local reference to it. If this is the case, the `ExceptionCheck` method is an inexpensive shortcut for `ExceptionOccurred` that does not return the exception itself. When a native method ends with a Java exception pending, the virtual machine discovers the exception and propagates it to the caller of the native method.

If you attempt to continue calling into the virtual machine while an exception is pending, the behavior is undefined.<sup>7</sup> Unfortunately, this leads to very cluttered code, with every `JNIEnv` call immediately followed by a check that no exception is pending, plus associated cleanup and recovery code if necessary, as seen in Listing 6–20.

#### Listing 6–20 Error-Safe Version of `JNI_OnLoad`

```
//Every JNIEnv* call is checked before continuing.
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* pEnv;
    if (JNI_OK != vm->GetEnv((void **)&pEnv, JNI_VERSION_1_2))
    {
```

---

7. If “undefined behavior” is starting to sound like a mantra, that is because it is one. A great achievement of Java is how many of its behaviors are well defined, even when problems occur. Because JNI is native code, it cannot guarantee well-defined behavior.

```

        return JNI_EVERSION;
    }
    jclass cls;
    if (NULL == (cls = pEnv->FindClass("NativePoint")))
        return JNI_EVERSION;
    if (NULL == (s_fldX = pEnv->GetFieldID(cls, "x", "I")))
        return JNI_EVERSION;
    if (NULL == (s_fldY = pEnv->GetFieldID(cls, "y", "I")))
        return JNI_EVERSION;
    return JNI_VERSION_1_2;
}

```

This is substantially more cluttered than the previous version, and frankly, this example still understates the general problem. All of the method calls above indicate failure by their return value, which is a little easier than calling `ExceptionOccurred` or `ExceptionCheck`. Also, this particular method required no cleanup in case of partial failure. A more complex JNI method would be even more cluttered with cleanup code.

The irony here is that this is the exact problem exceptions were designed to solve. JNI does not use C++ exceptions for simplicity and for backward compatibility with C. However, there is nothing preventing you from using C++ exceptions yourself, so long as you never let them propagate back into the virtual machine. The website for this book [Hal01] includes the `JNIEnvUtil` class, which is a plug-compatible subclass of `JNIEnv` that automates the process of converting a Java error into a C++ exception. For every JNI call that might fail, the `JNIEnvUtil` class calls back to `ExceptionOccurred` and then throws a C++ exception. For example, `CallVoidMethodA` looks like Listing 6–21.

#### Listing 6–21 JNIEnvUtil

```

struct JNIEnvUtil : public JNIEnv {
    void CallVoidMethodA(jobject obj, jmethodID methodID,
                        jvalue * args) {
        JNIEnv::CallVoidMethodA(obj, methodID, args);
        if (ExceptionOccurred()) {
            throw JNIException();
        }
    }
    //remainder omitted for clarity
}

```

The use of C++ exceptions allows you to structure your JNI code without worrying about virtual machine exceptions. If you were using `JNIEnvUtil`, the `JNI_OnLoad` method would look like Listing 6–22.

#### Listing 6–22 Using `JNIEnvUtil`

```
//JNIEnv replaced by JNIEnvUtil
JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnvUtil* pEnv;
    if (JNI_OK != vm->GetEnv((void**)&pEnv, JNI_VERSION_1_2)){
        return JNI_EVERSION;
    }
    try {
        jclass cls = pEnv->FindClass("NativePoint");
        s_fldX = pEnv->GetFieldID(cls, "x", "I");
        s_fldY = pEnv->GetFieldID(cls, "y", "I");
        return JNI_VERSION_1_2;
    }
    catch(const JNIException& exc) {
        //no need to "throw" anything, Java exception is pending
        return JNI_EVERSION;
    }
}
```

The code structure inside the `try` block is linear and easy to understand. Of course, this example assumes that the only reaction to a Java exception is to immediately return, allowing the exception to propagate back to the caller of the native method. More sophisticated use of C++ exceptions is possible; the `JNIEnvUtil` class simply provides a starting point. Regardless of what helper classes or macros you use to simplify JNI programming, there are two critical things to remember: Always handle Java exceptions before continuing to call through the `JNIEnv` pointer, and never allow C++ exceptions back into the virtual machine.

#### 6.5.4 Throwing Java Exceptions from Native Code

The last and smallest piece of the error-handling story is throwing your own exceptions back into the virtual machine. This is a simple matter. The `JNIEnv`



class includes helper functions that allow you to manually set the pending exception for the current virtual machine thread (as seen in Listing 6–23).

**Listing 6–23 Throwing Java exceptions from Native Code Struct JNIEnv**

```
struct JNIEnv {  
    jint Throw(jthrowable obj);  
    jint ThrowNew(jclass clazz, const char* msg);  
}; //remainder omitted for clarity
```

The `Throw` method sets a pending exception object, which you either caught earlier or created from scratch using the JNI construction APIs. The `ThrowNew` method is a shortcut that instantiates a pending exception and calls its single argument `String` constructor. You should use these methods in the same situations that you would choose a `throw` statement in ordinary Java code. Just remember that after you set a pending exception, you should do no more work with the `JNIEnv` before returning, unless you first handle the exception by calling `ExceptionClear`.

## 6.6 Resource Management

One of the most obvious differences between Java and C++ is the model for managing resources. In Java, you simply drop references to unused objects and trust the garbage collector to reclaim memory when necessary. In C++, you typically take explicit control of resource deallocation. The JNI boundary must provide a sensible mapping between these two programming styles. There are four interesting cases to consider:

1. How does native code communicate with the garbage collector to manage the lifetime of Java objects?
2. How does Java code manage the lifetime of native objects?
3. How does JNI handle arrays?
4. How does JNI handle strings?

Arrays are a special case because Java accesses and stores arrays in a way that is not necessarily compatible with the pointer-based access used in C++. Strings are a special case because Java usually uses the two-byte Unicode format for strings, while much existing C/C++ code uses one-byte ASCII or ANSI format.

### 6.6.1 Interacting with the Garbage Collector

The first problem has to do with the management of Java objects from native code. In Java, the virtual machine keeps track of all object references for you. When you compile an assignment statement in Java, it translates to a bytecode that the virtual machine recognizes as an assignment statement. However, the virtual machine has no way to recognize assignment statements that execute in native code. If you assign a `jobject` reference to another variable, the garbage collector will not know about the new reference, and it may relocate or reclaim the object. Listing 6–24 shows a dangerous assignment.

#### Listing 6–24 A Dangerous Assignment

```
static jobject rememberedPoint;
JNIEXPORT void JNICALL Java_NativePoint_nativeMove
    (JNIEnv *pEnv, jobject obj, jint xinc, jint yinc)
{
    rememberedPoint = obj; //BAD: GC MAY MOVE OR RECLAIM obj
}
```

You cannot simply store a `jobject` reference for use later. Unless specifically documented otherwise, `jobject` references in JNI are *local references*. Local references are only valid until a JNI method returns back to Java, and then only on the thread they rode in on.

The limited lifetime of local references is convenient for the garbage collector. When the virtual machine creates the argument stack for a JNI method, it marks each `jobject` as “currently in a native method,” preventing garbage collection from touching the object until the native method returns. After the method returns, the garbage collector is free to treat the object by its normal rules, reclaiming it if it is not referenced elsewhere.

Local reference lifetime is also convenient for JNI programmers because there is nothing for them to do. As long as you are content to use only the `jobjects` passed into the current method, you do not have to worry about explicit resource management. If you want to hold onto `jobject` references for longer periods of time, you must use the global reference APIs.

Global references give native code the ability to mark an object reference as “in use until further notice,” thus disabling the garbage collector’s ability to reclaim the object. To create a global reference, you call `NewGlobalRef`; to

delete a global reference, you call `DeleteGlobalRef`. Use global references to cache objects that you need to use later, in a different method invocation and/or on a different thread. Listing 6–25 demonstrates using a global reference to cache a class object that will be used to throw an exception.

#### Listing 6–25 Using Global References

```
static jfieldID s_fldX;
static jfieldID s_fldY;
static jclass clsIllegalArgExc;
JNIEXPORT void JNICALL Java_NativePoint_nativeMove
    (JNIEnv *pEnv, jobject obj, jint xinc, jint yinc)
{
    if ((xinc < 0) || (yinc < 0)) {
        pEnv->ThrowNew(clsIllegalArgExc, "increment less than zero");
    }
    int x = pEnv->GetIntField(obj, s_fldX);
    int y = pEnv->GetIntField(obj, s_fldY);
    pEnv->SetIntField(obj, s_fldX, x + xinc);
    pEnv->SetIntField(obj, s_fldY, y + yinc);
}

JNIEXPORT jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv* pEnv;
    if (JNI_OK != vm->GetEnv((void **)&pEnv, JNI_VERSION_1_2))
    {
        return JNI_EVERSION;
    }
    jclass cls = pEnv->FindClass("NativePoint");
    s_fldX = pEnv->GetFieldID(cls, "x", "I");
    s_fldY = pEnv->GetFieldID(cls, "y", "I");
    jclass temp = pEnv->FindClass(
        "java/lang/IllegalArgumentException");
    clsIllegalArgExc = (jclass) pEnv->NewGlobalRef(temp);
    pEnv->DeleteLocalRef(temp);
    return JNI_VERSION_1_2;
}

JNIEXPORT void JNICALL JNI_OnUnload(JavaVM* vm, void* reserved) {
    JNIEnv* pEnv;
    if (JNI_OK != vm->GetEnv((void **)&pEnv, JNI_VERSION_1_2))
    {
        return;
    }
    pEnv->DeleteGlobalRef(clsIllegalArgExc);
}
```

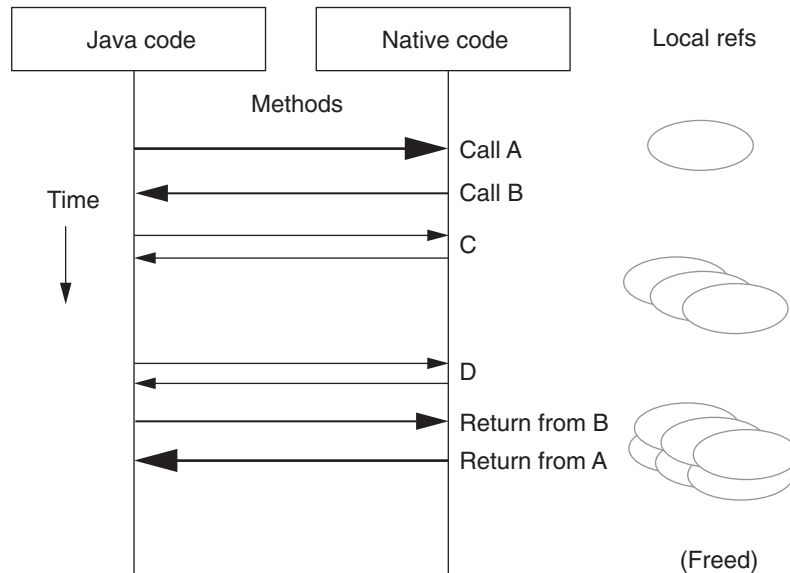
Listing 6–25 is similar to previous versions of `NativePoint`, except that this time around only positive movements are allowed; negative movements trigger an `IllegalArgumentException`. The `JNI_OnLoad` function calls `FindClass` to get the `jclass` reference for the `java.lang.IllegalArgumentException` class. However, the return value of `FindClass` is a local reference (remember that all JNI references are local unless explicitly documented otherwise). In order to safely cache the `jclass` reference for use later, you must call `NewGlobalRef` and cache the resulting global reference instead. When and if the native library is about to be unloaded, the `JNI_OnUnload` method will call `DeleteGlobalRef` to renounce its hold on the class object.

Managing global references is tricky because they add the complexities of C++ explicit memory management to Java references. In larger projects, it can be very difficult to identify where and when global references should be released. In fact, it can even be difficult to distinguish global from local references! Notice that global and local references both have the same static types, such as `jobject`, `jclass`, and so on. This means that the compiler cannot distinguish reference types. If you call `DeleteGlobalRef` on a local reference, the results are undefined. There is no silver bullet for these problems. However, they are standard fare for C++ developers, so at least writing JNI code is not substantially more difficult than any other kind of C++ programming.

Note that Listing 6–25 also includes a call to `DeleteLocalRef`. Local references expire at the end of the native method anyway, so `DeleteLocalRef` is not mandatory. However, there are two reasons that you might want to consider calling `DeleteLocalRef` as soon as you know that a reference is no longer needed by native code. First, the virtual machine may allocate only a limited number of local reference slots; by default only 16 are guaranteed to be available. By calling `DeleteLocalRef`, you give yourself the ability to do things such as iterating over large object arrays without exceeding the local capacity at any given time.

The second reason to call `DeleteLocalRef` is to guarantee that resources are reclaimed in a timely manner. While the JNI specification claims that local references are invalidated at the end of a native method, the virtual machine is not

always aggressive in reclaiming these references. Some virtual machines may not reclaim any local references until there are *no* native methods active on a thread, as shown in Figure 6–3.



Refs may not be freed until outermost native method A returns.

**Figure 6–3 Local references freed after native outermost call returns**

It is very important to call `DeleteLocalRef` when you are writing helper functions that may be called from other native methods because you have no idea how many reference slots will be available when the function executes. Expediently calling `DeleteLocalRef` can cause dramatic performance improvements in some situations. Of course, this advice depends on JNI implementation details, which can vary from one virtual machine to another. Always profile your code to be sure.

If you need more than the default 16 local references, the SDK 1.2 version of JNI provides APIs for reserving additional capacity and for allocating additional references, as shown in Listing 6–26.

**Listing 6-26 JNI Local Reference APIs**

```

struct JNIEnv {
    jint EnsureLocalCapacity(jint capacity);
    jobject NewLocalRef(jobject ref);
    //remainder omitted for clarity
};

```

You can call `EnsureLocalCapacity` at any time to set aside storage for a number of local references, which you can later use with any API that returns a new local reference, such as `NewLocalRef`. `EnsureLocalCapacity` returns zero to indicate success; otherwise, it returns a negative number and sets a pending `OutOfMemoryError`. Listing 6-27 shows how you can use `EnsureLocalCapacity` to set aside enough local reference slots to process an entire array.

**Listing 6-27 Using EnsureLocalCapacity**

```

JNIEXPORT jobject JNICALL Java_NativePoint_findBestMatch
    (JNIEnv* pEnv, jobject pt, jobjectArray ptArray)
{
    jsize size = pEnv->GetArrayLength(ptArray);
    pEnv->EnsureLocalCapacity(size);
    jobject* pts = new jobject[size];
    int n;
    for (n=0; n<size; n++) {
        pts[n] = pEnv->GetObjectArrayElement(ptArray, n);
    }
    int xSearch = pEnv->GetIntField(pt, s_fldX);
    jobject result;
    for (n=0; n<size; n++) {
        jobject objCur = pts[n];
        int xCur = pEnv->GetIntField(objCur, s_fldX);
        if (xCur == xSearch) {
            result = pEnv->NewLocalRef(objCur);
            break;
        }
    }
    for (n=0; n<size; n++) {
        pEnv->DeleteLocalRef(pts[n]);
    }
    delete [] pts;
    return result;
}

```

The `findBestMatch` method first guarantees that it has enough local references with `EnsureLocalCapacity`, then it extracts all the `jobjects` into an array, and then it scans that array looking for the `NativePoint` with the exact same x coordinate as the `pt` reference, a.k.a. `this`. To reclaim references as quickly as possible, the method loops over the array again, calling `DeleteLocalRef`. Note that this example did not strictly require `EnsureLocalCapacity`. This particular array traversal could have been done one local reference at a time, but more complex traversals might require simultaneous access to several elements.

If you are working with a block of references that will all go out of scope together, it is more convenient to use the `PushLocalFrame` and `PopLocalFrame` APIs shown in Listing 6–28.

#### Listing 6–28 Managing Reference Frames

```
struct JNIEnv {
    jint PushLocalFrame(jint capacity);
    jobject PopLocalFrame(jobject result);
    //remainder omitted for clarity
};
```

`PushLocalFrame` creates a new frame, and the next `capacity` references will belong to it. Similar to `EnsureLocalCapacity`, `PushLocalFrame` either returns zero on success, or it returns a negative number and sets a pending `OutOfMemory` error on failure. These references can then all be released in one motion with `PopLocalFrame`, which also allows one reference to be kept alive as a logical return value for the frame. These APIs make the `findBestMatch` method much cleaner, as shown in Listing 6–29.

#### Listing 6–29 Using Reference Frames

```
JNIEXPORT jobject JNICALL Java_NativePoint_findBestMatch
(JNIEnv* pEnv, jobject pt, jobjectArray ptArray)
{
    jsize size = pEnv->GetArrayLength(ptArray);
    pEnv->PushLocalFrame(size);
    jobject* pts = new jobject[size];
    int n;
    for (n=0; n<size; n++) {
```

```

        pts[n] = pEnv->GetObjectArrayElement(ptArray, n);
    }
    int xSearch = pEnv->GetIntField(pt, s_fldX);
    jobject result;
    for (n=size-1; n>=0; n--) {
        jobject objCur = pts[n];
        int xCur = pEnv->GetIntField(objCur, s_fldX);
        if (xCur == xSearch) {
            result = pEnv->PopLocalFrame(objCur);
            break;
        }
    }
    delete [] pts;
    return result;
}

```

Even if you never call `EnsureLocalCapacity` or `PushLocalFrame`, you may be able to create all the local references you want. The JNI documentation states that “For backward compatibility, the VM allocates local references beyond the ensured capacity.”<sup>8</sup> In my tests of the 1.2 Classic and 1.3 HotSpot implementations of the virtual machine on Windows NT 4.0, I found that it is possible to create tens of thousands of local references without any complaint from the virtual machine. The `-verbose:jni` command-line option for the `java` launcher is supposed to provide warning messages if you exceed your local reference capacity, but this works only sometimes. On the classic VM, the warnings appear to work correctly when you reserve small numbers of local references, up to about a thousand. However, if you `Ensure` space for a large number of objects, the call returns successfully, but erroneous warning messages begin to appear claiming that you have exceeded the 16 reference limit. Furthermore, the warning flags appear to have been dropped entirely from the HotSpot VM.

The point of this digression is *not* to encourage you to rely on these idiosyncrasies of the Windows VM implementation, but quite the opposite. Instead, you should be aware that the virtual machine does not reliably warn you if you misuse the local reference APIs. Be careful to follow the local reference rules and

---

8. See the JNI specification [Lia99] section on `EnsureLocalCapacity`.





program to the specification, not to the current behavior of a particular virtual machine. This will give your code the best chance of being portable to a wide variety of virtual machines and operating systems.

### 6.6.2 Managing Native Resources

There is not much to say about managing native resources from Java because JNI does not define any Java mapping for native objects. You must handroll any resource management scheme you wish to use. Fortunately, the core API has several examples of how to do this. Sockets, files, and database connections are all native resources that are hidden behind Java objects. With each of these native resources, the idiom is the same: At the native API level, there are function calls you can use to allocate the resource and return it to the pool. These function calls must be mapped to appropriate methods on a Java object. For resource allocation, the Java constructor is an obvious match, as shown here:

```
public class NativeResource {
    private int handle;
    private native int allocHandle();
    public NativeResource() {
        handle = allocHandle();
    }
}
```

You cannot use the OS call directly because it does not match the signature expected by JNI, so `allocHandle` maps the JNI signature to the OS function call instead:

```
JNIEXPORT jobject JNICALL NativeResource_allocHandle
(JNIEnv* pEnv, jobject pt)
{
    return OSAllocResource(); //placeholder for some OS call
}
```

Deallocating the resource is a bit more of a challenge. If you were wrapping an API in C++, you could place the call to `OSDeallocResource` in the class destructor. However, Java does not have destructors. Instead, most Java classes take a two-pronged approach to freeing native resources: Define a



`close` method that deallocates any native resources, and then back that up with a `finalize` method that calls `close` automatically, as shown in Listing 6–30.

### Listing 6–30 Managing Native Resources

```
//continuing class NativeResource
private native deallocHandle(int handle);
public void close() {
    if (handle != 0) {
        deallocHandle(handle);
        handle = 0;
    }
}
protected void finalize() {
    close();
}
public useResource(String someArg) {
    if (handle == 0) {
        throw new IllegalStateException("object is closed");
    }
    //do work...
}
```

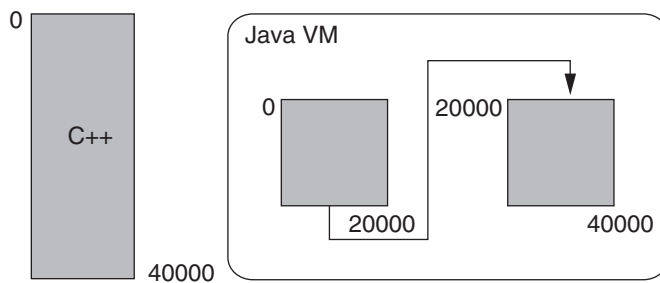
This solution introduces a fair amount of complexity. Since the class is not safe for use once the native resource is closed, all the actual functionality of the class must be guarded by `if (handle == 0)` blocks, as the `useResource` method demonstrates. Clients of the class must be encouraged to call `close` as soon as they know they are finished with the resource so that the resource can be reclaimed as quickly as possible. Clients may even want to call `close` within a `finally` block to guarantee that the object is closed even in exceptional situations.

If a client forgets to call `close`, you can only hope that the `finalize` method will be called soon. However, `finalize` is neither reliable nor efficient in helping reclaim native resources. The simple fact is this: Garbage collection is often a great solution for memory management, but it does not help you with non-memory resource management. Native resources need to be explicitly deallocated, just as they would be in a systems programming language such as C++. If you write a Java class that wraps a native resource, use the `close/finalize` tandem to quickly reclaim resources that are no longer needed.

### 6.6.3 Managing Arrays

The third problem area that JNI faces is passing arrays. In C++, an array is a contiguous block of memory that can be addressed by a pointer. So, assuming that an int is four bytes long, an array of 10,000 ints would be stored as 40,000 contiguous bytes somewhere in memory. Using pointer math, it is trivial to address any particular element in the array. If the array begins at `0x55551000`, then the fourth element of the array is at `0x55551010`.

The Java definition of an array is looser. Java does not expose pointers to the programmer, and array accesses in Java are actual bytecode instructions processed by the virtual machine. Since the VM is queried for each item in the array, it does not need to store the array contiguously in memory. As long as the VM is willing to do the bookkeeping, it can choose to store the array as several smaller pieces. This is entirely transparent to the Java programmer, of course. Figure 6–4 demonstrates possible C++ and Java storage of the same array.



**Figure 6–4 Contiguous versus fragmented storage for a byte array**

This difference in storing arrays becomes a problem when you are passing an array between Java and native code. A C++ programmer cannot simply take the raw pointer address of a Java array and then index to a particular element because that part of the array might be elsewhere in memory. JNI provides three related API sets to address this problem. All of the APIs do basically the same thing: They take a Java array and convert it into the format expected by a C++ programmer, and vice versa.

The simplest array API functions have the word `Region` in their names, and they simply copy pieces of arrays between C++ and Java formats. The `Region` APIs are shown in Listing 6–31.

#### Listing 6–31 JNI Array Region API

```
struct JNIEnv {
    // Only the int version is shown; there are similar calls
    //for boolean, byte, char, short, long, float, and double
    void GetIntArrayRegion(jintArray arr, jsize start,
                          jsize len, jint* buf);
    void SetIntArrayRegion(jintArray arr, jsize start,
                          jsize len, jint* buf);
    //remainder omitted for clarity
};
```

Because the `Region` API functions always create a new copy of the data they manipulate, you do not have to worry about how the source or destination arrays are stored in memory. As a result, these functions are the easiest to understand. Listing 6–32 shows how to use the `Region` API to increment each element in a Java array.

#### Listing 6–32 Using the Region API

```
JNIEXPORT void JNICALL Java_TestNativeArray_incByRegionAPI
(JNIEnv *pEnv, jclass cls, jintArray arr, jint inc)
{
    jint size = pEnv->GetArrayLength(arr);
    jint* carray = new jint[size];
    //copy initial Java array into contiguous C array
    pEnv->GetIntArrayRegion(arr, 0, size, carray);
    for (jint n=0; n<size; n++) {
        carray[n] += inc;
    }
    //copy result elements back into Java array
    pEnv->SetIntArrayRegion(arr, 0, size, carray);
    delete [] carray;
}
```

Because the array is copied into a C-style array, you must allocate an array to receive the contents, as was done here with C++ `new` and `delete`. The logic is simple but potentially very wasteful. The entire array is copied twice, once into

C++ contiguous memory, and again back into whatever storage the virtual machine is using. For large arrays, this copying overhead can easily dwarf the work being done. If it does, you should consider using one of the two other array APIs, which do not mandate a copy every time.

The second array API is the `Element` API (see Listing 6–33). Again, there are API functions for each primitive type, and for simplicity, the text will always refer to the `int` version. Unlike the `Region` APIs, the `Element` APIs will attempt to give native code a direct pointer to the array elements; however, this may not always be possible. If the virtual machine keeps the array in contiguous memory, it may give a direct pointer to native code. If the Java array is not contiguous, then the virtual machine will have to copy the array to a contiguous block for use from native code.

#### Listing 6–33 JNI Array Element API

```
struct JNIEnv {
    //element APIs. Only the int version is shown; there are
    //calls for boolean, byte, char, short, long, float, double
    jint* GetIntArrayElements(jintArray arr, jboolean* isCopy);
    void ReleaseIntArrayElements(jintArray arr, jint* elems,
                                jint mode);
};
```

The Java Language Specification does not provide any technique to guarantee how an array is stored, or to query whether an array is stored contiguously. Even if the array is stored contiguously, the virtual machine could still choose to copy the array when it was making it accessible to native code. When you use the `Element` APIs, you must write code that functions correctly regardless of whether the array is being copied or modified in place.

This complicates the programming model in several ways. Since you do not know whether the array will be copied, you cannot allocate native memory in advance. If the `Element` API decides to copy the array, it will allocate the memory internally, and you must use the corresponding `ReleaseTypeArrayElements` function to deallocate the memory. This contrasts with the `Region` APIs discussed above, where it is perfectly acceptable to call `GetIntArrayRegion` without ever making a corresponding call to `SetIntArrayRegion`. Listing 6–34 reimplements the prior example, shown in Listing 6–32, this time using the `Element` APIs.

**Listing 6–34 Using the Array Element API**

```

JNIEXPORT void JNICALL Java_TestNativeArray_incByPinning
    (JNIEnv *pEnv, jclass cls, jintArray arr, jint inc)
{
    jint size = pEnv->GetArrayLength(arr);
    //pass zero because code does not depend on whether array
    //      is copied
    jint* carray = pEnv->GetIntArrayElements(arr, 0);
    for (jint n=0; n<size; n++) {
        carray[n] += inc;
    }
    pEnv->ReleaseIntArrayElements(arr, carray, 0);
}

```

When you call `GetIntArrayElements`, you pass in a `jboolean*` that will record whether you actually have direct access to the array. If the array is copied, then you must call `ReleaseIntArrayElements` to copy your changes back to the master version of the array and to deallocate the temporary copy of the array. Even if you are directly manipulating the master copy of the array, you still need to call `ReleaseIntArrayElements`, because of array *pinning*.

Pinning prevents the garbage collector from moving the array to another location while you are using it. Moving memory out from under a native pointer would cause undefined behaviors that are very difficult to debug, so the virtual machine marks the array as unmoveable, preventing it from moving until you call `ReleaseIntArrayElements`. In order for `ReleaseIntArrayElements` to give precise control over the pinning process, it takes a `mode` flag that controls unpinning the array. The modes work differently for copied versus pinned arrays, as shown in Table 6–4.

The good news is that you rarely need to write special case code based on whether an array was copied or pinned. The excitingly named `0` flag is generally appropriate for read/write traversal of an array. If the array was copied, it copies the temporary array back into the master copy, and if the array was pinned, it unpins it.

The `JNI_ABORT` flag supports read-only traversal. If the array was copied, `JNI_ABORT` does not bother to copy the array back, which improves performance. Note that the `JNI_ABORT` flag does not function as a transactional roll-

**Table 6–4 Array Unpin Modes**

Mode Flag	Meaning	Effect on Copied Array	Effect on Pinned Array	Usage
0	Done, post changes	Copies back to master, frees temp array	Unpins	Read/write traversal
JNI_ABORT	Done, drop changes	Frees temp array	Unpins, <i>changes kept anyway!</i>	Read-only traversal
JNI_COMMIT	Not done, post changes	Copies back to master	No effect	Rarely used

back. If the changes were made directly to the pinned array, `JNI_ABORT` will not unmake the changes. `JNI_NOCOPY` would probably be a better name for the `JNI_ABORT` flag.

The `JNI_COMMIT` flag allows you to copy changes back without releasing the temporary array. This is useful in rare circumstances in which you are making many changes to an array and want those changes to be quickly visible to another thread.<sup>9</sup>

Some virtual machines do not support pinning and always return a copy of the array to the `GetIntArrayElements` call. In order to give such virtual machines another performance option with large arrays, SDK 1.2 introduced a third array access API called the `Critical` API, which is shown in Listing 6–35.

#### Listing 6–35 JNI Array Critical API

```
struct JNIEnv {
//Unlike the other APIs, there are no
//typed versions for each primitive type, just void*.
void* GetPrimitiveArrayCritical(jarray array,
                               jboolean* isCopy);
void ReleasePrimitiveArrayCritical(jarray array,
                                   void* carr,
                                   int mode);

//remainder omitted for clarity
};
```

9. To make this work, you must also correctly use synchronized blocks in Java and the `MonitorEnter` and `MonitorExit` functions in JNI. Interesting topics in themselves, but outside the scope of this book.

The `Critical` API is very similar to the `Elements` API discussed previously. `GetPrimitiveArrayCritical` may choose to copy or pin, and `ReleasePrimitiveArrayCritical` uses the same control flags that were described earlier. The key difference is that the `Critical` APIs are more likely to provide direct access because they do not require pinning. However, the better odds come at a price. After calling `GetPrimitiveArrayCritical`, you enter a *critical region*. Until you call `Release`, you must not call other JNI functions, block the thread at the native or Java level, or take very much time. These restrictions allow a virtual machine to employ simple, draconian means for protecting the array while you access it (for example, it may *stop all other VM threads*). Listing 6–36 shows the `Critical` array APIs being used.

#### Listing 6–36 Using the Critical Array APIs

```
JNIEXPORT void JNICALL Java_TestNativeArray_incCritical
    (JNIEnv *pEnv, jclass cls, jintArray arr, jint inc)
{
    jint size = pEnv->GetArrayLength(arr);
    //pass zero because code does not depend on whether array
    //      is copied
    jint* carray = (jint*)
        pEnv->GetPrimitiveArrayCritical(arr, 0);
    //BEGIN CRITICAL SECTION
    for (jint n=0; n<size; n++) {
        carray[n] += inc;
    }
    //END CRITICAL SECTION
    pEnv->ReleasePrimitiveArrayCritical(arr, carray, 0);
}
```

After taking the extra care required to use the `Critical` APIs, you may discover that their impacts on overall performance are positive, negative, or indifferent. If the speed gain from direct access to the array outweighs the cost of stopping other threads, then the `Critical` API is a better choice than the `Elements` API, but the reverse can also be true. If the virtual machine supports pinning arrays, or if all of the APIs are forced to copy arrays, you may discover no performance effects at all.



If you reach a point where the performance of the JNI array APIs is significant to your application, you will need to profile so that you can choose between the `Region`, `Elements`, and `Critical` APIs. Moreover, you will need to profile on the variety of operating systems, virtual machines, and hardware you intend to support. This is true throughout Java, but it is especially true with JNI, which may vary widely from one operating system to another. Write once, run anywhere, profile everywhere.

#### 6.6.4 Managing Strings

Strings create many of the same problems that arrays do, which is not surprising since strings are basically arrays with some additional semantics attached. As you can see in Listing 6–37, JNI provides the same three API families for strings that it does for arrays. As with arrays, the `Region` functions always copy the string into a preallocated buffer. The `Chars` API for strings corresponds to the `Element` array API, and it will try to pin the string in memory, otherwise copying it. The `Critical` string API attempts to provide direct access to the string without pinning it. The `Critical` API functions by basically shutting down the rest of the virtual machine, and its use is governed by the same rules listed for the `Critical` array API: no JNI callbacks, no thread blocking, and quick execution.

##### Listing 6–37 JNI String API

```
struct JNIEnv {
    //basics, available since 1.1
    jstring NewString(const jchar *unicode, jsize len);
    jsize GetStringLength(jstring str);
    jstring NewStringUTF(const char *utf);
    jsize GetStringUTFLength(jstring str);

    //region API, available since 1.2
    void GetStringRegion(jstring str, jsize start,
                        jsize len, jchar *buf);
    void GetStringUTFRegion(jstring str, jsize start,
                           jsize len, char *buf);

    //pinning strings, available since 1.1
    const jchar* GetStringChars(jstring str, jboolean *isCopy);
```

```

void ReleaseStringChars(jstring str, const jchar *chars);
const char* GetStringUTFChars(jstring str,
                               jboolean *isCopy);
void ReleaseStringUTFChars(jstring str, const char* chars);

//critical API, available since 1.2
const jchar * GetStringCritical(jstring string,
                                jboolean *isCopy);
void ReleaseStringCritical(jstring string,
                           const jchar *cstring);
//remainder omitted for clarity
}

```

There are two special characteristics of Java `Strings` that deserve attention. First, Java `Strings` are immutable—that is, they cannot be changed once they are instantiated. This greatly simplifies the various APIs. `Strings` have `GetRegion` but no `SetRegion` because it would be illegal to copy changed characters into a `String`. Similarly, the `Chars` and `Critical` APIs do not have a flag to control unpinning behavior. There is no behavior to control because changes can never be copied back.

Second, Java strings are likely stored internally as Unicode. The JNI string API comes in two flavors. The flavor that uses the UTF acronym in its name and takes `char*` parameters converts Java strings into UTF-8 format,<sup>10</sup> which is a single-byte format that is compatible with ASCII in the lower seven bits. The other flavor takes `jchar` arguments and works with two-byte Unicode strings. Since virtual machines tend to use Unicode internally, it is a good bet that the UTF functions will always return copies, while the Unicode functions have some chance of providing direct access. Unfortunately, most legacy C++ code uses single-byte string encoding, so you will likely find yourself using the UTF APIs.

## 6.7 Onward

Very few software systems build on only one technology, so interoperability is key to the success of large projects. JNI provides a low-level, in-process model for interoperation between Java and systems code written in C or C++. Studying

---

10. Java's UTF-8 format is slightly different from the standard UTF-8 format. The Java format is documented in [LY99].

JNI is a valuable way to learn about both C++ and Java because JNI has to deal with how the language worlds differ. JNI provides a mapping from Java types to a dynamic, metadata-driven, handle-based, C-callable API.

JNI provides a dynamic loading architecture for native code that is a stripped-down version of Java's powerful class loader architecture. JNI provides APIs to make Java exceptions, arrays, and strings usable from native code, and it copes with the fact that these constructs do not naturally map to their C++ counterparts. JNI provides an API for explicitly managing Java object lifetime, which is necessary when you need long-lived "global" references that are safe from garbage collection.

Unfortunately, JNI is too low-level to be the perfect solution. Frankly, it is the assembly language of interop. JNI makes no attempt to map C/C++ objects into Java. Nor does it provide tools to automate wrapping existing native code libraries; instead, you have to manually write wrapper functions that translate from JNI signatures to your existing native code APIs. If anything, this chapter should convince you to avoid JNI wherever possible.

The Java world needs a higher-level approach for interoperating with other component platforms. Many attempts have been made and some have become commercially viable, but none have become part of Java. Appendix A describes an open-source library for interoperation with Win32 and COM components, and [JavaWin32] lists the various interop products.

## 6.8 Resources

[Lia99] is a well-written guide from a designer of JNI; you will find that Chapters 9 and 10 of this guide are particularly valuable. [Lia99] also includes the JNI specification, which is clear and concise. The only real complaint you may have is that the Java 2 SDK enhancements are in a separate add-on document, so you have to guess when a feature was implemented in order to find its documentation.

As mentioned previously, [JavaWin32] lists interop products that provide enhancements over raw JNI for calling between Java and native code on Windows operating systems.

