Best practices for using the Java Native Interface

Techniques and tools for averting the 10 most common JNI programming mistakes

Michael Dawson
Advisory Software Developer
IBM

Graeme Johnson
J9 Virtual Machine Development Manager
IBM

Andrew Low
STSM, J9 Virtual Machine
IBM

07 July 2009

The Java™ Native Interface (JNI) is a standard Java API that enables Java code to integrate with code written in other programming languages. JNI can be a key element in your toolkit if you want to leverage existing code assets — for example, in a service-oriented architecture (SOA) or a cloud-based system. But when used without due care, JNI can quickly lead to poorly performing and unstable applications. This article identifies the top 10 JNI programming pitfalls, provides best practices for avoiding them, and introduces the tools available for implementing these practices.

The Java environment and language are safe and efficient for application development. However, some applications need to perform tasks that go beyond what can be done from within a pure-Java program, such as:

JNI's evolution

JNI has been part of the Java platform since the JDK 1.1 release and was extended in the JDK 1.2 release. The JDK 1.0 release included an earlier native-method interface that lacked clean separation between native and Java code. In this interface, natives would reach directly into JVM structures and so could not be portable across JVM implementations, platforms, or even versions of the JDK. Upgrading an application with a substantial number of natives using the JDK 1.0 model was expensive, as was developing natives that could run with multiple JVM implementations.

The introduction of JNI in the JDK 1.1 release allowed:

- Version independence
- Platform independence
- VM independence
- Development of third-party class libraries

It is interesting to note that younger languages such as PHP are still struggling with these issues with respect to their support of native code.

- Integrate with existing legacy code to avoid a rewrite.
- Implement functionality missing in available class libraries. For example, you might need Internet Control Message Protocol (ICMP) functionality if you are implementing `ping` in the Java language, but the base class libraries don't provide it.
- Integrate with code that's best written in C/C++, to exploit performance or other environment-specific system characteristics.
- Address special circumstances that require non-Java code. For example, implementation of core class libraries might require cross-package calls or the need to bypass other Java security checks.

The JNI lets you accomplish these tasks. It provides a clean separation between the execution of Java code and native code (such as C/C++) by defining a clear API for communicating between the two. For the most part, it avoids direct memory reference by native code into the JVM, ensuring that natives can be written once and work across different JVM implementations or versions.

With JNI, native code is free to interact with Java objects, get and set field values, and invoke methods without many of the constraints that apply to the same functions in Java code. This freedom is a double-edged sword: it trades the safety of the Java language for the ability to accomplish the tasks listed earlier. Using JNI within your application provides powerful low-level access to the machine resources (memory, I/O, and so on), so you're working without the safety net usually provided to Java developers. JNI's flexibility and power introduce the risk of programming practices that can lead to poor performance, bugs, and even program crashes. You must be careful about the code you include in your application and use good practices to safeguard the application's overall integrity.

Develop skills on this topic

This content is part of a progressive knowledge path for advancing your skills. See Using the Java Native Interface

This article covers the 10 most common coding and design errors that users of the JNI make. The goal is to help you recognize and steer clear of them so that you can write safe, effective JNI code that performs well from the start. This article also introduces available tools and techniques for finding these issues in new or existing code and show how to apply them effectively.

JNI programming pitfalls fall into two categories:

- **Performance**: The code performs the designed function but does so slowly or in a way that causes the overall program to slow down.

- **Correctness**: The code works some of the time but does not reliably provide the required function; in the worst case, it crashes or hangs.

Performance pitfalls

The top five performance pitfalls for programmers using JNI are:

- Not caching method IDs, field IDs, and classes
- Triggering array copies
- Reaching back instead of passing parameters
- Choosing the wrong boundary between native and Java code
- Using many local references without informing the JVM

Not caching method IDs, field IDs, and classes

To access Java objects' fields and invoke their methods, native code must make calls to `FindClass()`, `GetFieldID()`, `GetMethodId()`, and `GetStaticMethodID()`. In the case of `GetFieldID()`, `GetMethodID()`, and `GetStaticMethodID()`, the IDs returned for a given class don't change for the lifetime of the JVM process. But the call to get the field or method can require significant work in the JVM, because fields and methods might have been inherited from superclasses, making the JVM walk up the class hierarchy to find them. Because the IDs are the same for a given class, you should look them up once and then reuse them. Similarly, looking up class objects can be expensive, so they should be cached as well.

For example, Listing 1 shows the JNI code required to call a static method:

Listing 1. Calling a static method with JNI

```
int val=1;
jmethodID method;
jclass cls;

cls = (*env)->FindClass(env, "com/ibm/example/TestClass");
if ((*env)->ExceptionCheck(env)) {
   return ERR_FIND_CLASS_FAILED;
}
method = (*env)->GetStaticMethodID(env, cls, "setInfo", "(I)V");
if ((*env)->ExceptionCheck(env)) {
   return ERR_GET_STATIC_METHOD_FAILED;
}
(*env)->CallStaticVoidMethod(env, cls, method,val);
if ((*env)->ExceptionCheck(env)) {
   return ERR_CALL_STATIC_METHOD_FAILED;
}
```

Looking up the class and method ID every time we want to call the method results in six native calls instead of the two that would be required if we had cached the class and method ID the first time they were needed.

Caching makes a significant impact on your application's run time. Consider the following two versions of a method, which end up doing the same thing. The version in Listing 2 uses cached field IDs:

Listing 2. Using cached field IDs

```
int sumValues2(JNIEnv* env, jobject obj, jobject allValues){

   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);

   return avalue + bvalue + cvalue + dvalue + evalue + fvalue;
}
```

Performance Tip #1

Look up and globally cache commonly used classes, field IDs, and method IDs.

Listing 3 doesn't use cached field IDs:

Listing 3. Field IDs not cached

```
int sumValues2(JNIEnv* env, jobject obj, jobject allValues){
   jclass cls = (*env)->GetObjectClass(env,allValues);
   jfieldID a = (*env)->GetFieldID(env, cls, "a", "I");
   jfieldID b = (*env)->GetFieldID(env, cls, "b", "I");
   jfieldID c = (*env)->GetFieldID(env, cls, "c", "I");
   jfieldID d = (*env)->GetFieldID(env, cls, "d", "I");
   jfieldID e = (*env)->GetFieldID(env, cls, "e", "I");
   jfieldID f = (*env)->GetFieldID(env, cls, "f", "I");
   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);
   return avalue + bvalue + cvalue + dvalue + evalue + fvalue
}
```

The version in Listing 2 takes 3,572 ms to run 10,000,000 times. Listing 3's version takes 86,217 ms — 24 times longer.

Triggering array copies

JNI provides a clean interface between Java code and native code. To maintain this separation, arrays are passed as opaque handles, and native code must call back to the JVM in order to manipulate array elements using set and get calls. The Java specification leaves it up to the JVM implementation whether these calls provide direct access to the arrays or return a copy of the array. For example, the JVM might return a copy when it has optimized arrays in a way that does not store them contiguously. (See Resources for a description of one such JVM.)

These calls, then, might cause copying of the elements being manipulated. For example, if you call `GetLongArrayElements()` on an array with 1,000 elements, you might cause the allocation and copy of at least 8,000 bytes (1,000 elements * 8 bytes for each `long`). When you then update the array's contents with `ReleaseLongArrayElements()`, another copy of 8,000 bytes might be

required to update the array. Even when you use the newer `GetPrimitiveArrayCritical()`, the specification still permits the JVM to make copies of the full array.

Performance Tip #2

Get and update only those parts of an array that the native needs. Use the appropriate API calls to avoid copying the whole array when only part of it is needed.

The `Get`*`Type`*`ArrayRegion()` and `Set`*`Type`*`ArrayRegion()` methods allow you to get and update a region of an array, as opposed to the full array. By using these methods to access larger arrays, you can ensure that you copy only the region of the array that the native will actually use.

For example, consider two versions of the same method, shown in Listing 4:

Listing 4. Two versions of the same method

```
jlong getElement(JNIEnv* env, jobject obj, jlongArray arr_j,
                 int element){
   jboolean isCopy;
   jlong result;
   jlong* buffer_j = (*env)->GetLongArrayElements(env, arr_j, &isCopy);
   result = buffer_j[element];
   (*env)->ReleaseLongArrayElements(env, arr_j, buffer_j, 0);
   return result;
}

jlong getElement2(JNIEnv* env, jobject obj, jlongArray arr_j,
                  int element){
     jlong result;
     (*env)->GetLongArrayRegion(env, arr_j, element,1, &result);
     return result;
}
```

The first version might cause two full copies of the array, whereas the second causes no copying at all. Running the first method 10,000,000 times with an array of 1,000 bytes takes 12,055 ms; the second version takes only 1,421 ms. The first version takes 8.5 times longer!

Performance Tip #3

Get or update as much of an array as possible in a single API call. Don't iterate through the array's elements one by one when you can get and update an array in larger blocks.

On the other hand, using `Get`*`Type`*`ArrayRegion()` to get each of the array's elements one by one will also not perform well if you're going to end up getting all of the array's elements anyway. For best performance, ensure that you get and update array elements in the largest sensible blocks. If you're going to iterate through all of the elements in an array, neither of the two `getElement()` methods in Listing 4 is suitable. Instead, you'd want to get a reasonable-sized chunk of the array in one call and then iterate through all of those elements, repeating until you cover the full array.

Reaching back instead of passing parameters

When calling a method, you can often choose between passing a single object that has multiple fields and passing the fields individually. With object-oriented designs, passing the object often provides better encapsulation, in that changes in the object fields don't require changes to the

method signature. However, in the case of JNI, a native must reach back into the JVM through one or more JNI calls to get the value for each individual field that it needs. These additional calls add extra overhead because the transition from native to Java code is more expensive than a normal method call. For JNI, therefore, causing the natives to reach for many individual fields from the objects passed to them leads to poorer performance.

Consider two methods in Listing 5, the second of which assumes we have cached the field IDs:

Listing 5. Two method versions

```
int sumValues(JNIEnv* env, jobject obj, jint a, jint b,jint c, jint d, jint e, jint f){
   return a + b + c + d + e + f;
}

int sumValues2(JNIEnv* env, jobject obj, jobject allValues){

   jint avalue = (*env)->GetIntField(env, allValues, a);
   jint bvalue = (*env)->GetIntField(env, allValues, b);
   jint cvalue = (*env)->GetIntField(env, allValues, c);
   jint dvalue = (*env)->GetIntField(env, allValues, d);
   jint evalue = (*env)->GetIntField(env, allValues, e);
   jint fvalue = (*env)->GetIntField(env, allValues, f);

   return avalue + bvalue + cvalue + dvalue + evalue + fvalue;
}
```

Performance Tip #4

When possible, pass individual parameters to JNI natives so that the native calls back to the JVM for the data it needs to do its work.

The `sumValues2()` method requires six JNI callbacks and takes 3,572 ms to run 10,000,000 times. It is six times slower than `sumValues()`, which takes only 596 ms. By passing in the data required by the JNI method, `sumValues()` avoids a substantial amount of JNI overhead.

Choosing the wrong boundary between native and Java code

It's up to the developer to define the boundary between native and Java code. The choice of the boundary can have a significant impact on the application's overall performance. The cost of calling from Java code to natives and from natives to Java code is significantly higher than a normal Java method call. Further, the transition can interfere with the JVM's ability to optimize code execution. For example, the Just-in-time compiler might be less effective as the number of transitions between Java code and native code increases. We have measured that calling from Java code to a native can take five times longer than a regular method. Similarly, calls from a native to Java code can take substantial time.

Performance Tip #5

Define the split between Java and natives to minimize the transitions from Java to natives and callbacks from natives to Java.

The split between Java code and natives should, therefore, be designed to minimize the transitions between Java and native code. Transitions should be made only when required, and you should do enough work in a native to amortize the cost of the transition. A key element of minimizing

transitions is to ensure that data is maintained on the correct side of the Java/native boundary. If data resides on the wrong side, constant transitions will be triggered by the need of the other side to reach for that data.

For example, if we want to provide an interface to a serial port using JNI, we could come up with two different interfaces. One version is in Listing 6:

Listing 6. Interface to a serial port: Version 1

```
/**
 * Initializes the serial port and returns a java SerialPortConfig objects
 * that contains the hardware address for the serial port, and holds
 * information needed by the serial port such as the next buffer
 * to write data into
 *
 * @param env JNI env that can be used by the method
 * @param comPortName the name of the serial port
 * @returns SerialPortConfig object to be passed ot setSerialPortBit
 *          and getSerialPortBit calls
 */
jobject initializeSerialPort(JNIEnv* env, jobject obj,  jstring comPortName);

/**
 * Sets a single bit in an 8 bit byte to be sent by the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig object returned by initializeSerialPort
 * @param whichBit value from 1-8 indicating which bit to set
 * @param bitValue 0th bit contains bit value to be set
 */
void setSerialPortBit(JNIEnv* env, jobject obj, jobject serialPortConfig,
  jint whichBit,  jint bitValue);

/**
 * Gets a single bit in an 8 bit byte read from the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig object returned by initializeSerialPort
 * @param whichBit value from 1-8 indicating which bit to read
 * @returns the bit read in the 0th bit of the jint
 */
jint getSerialPortBit(JNIEnv* env, jobject obj, jobject serialPortConfig,
  jint whichBit);

/**
 * Read the next byte from the serial port
 *
 * @param env JNI env that can be used by the method
 */
void readNextByte(JNIEnv* env, jobject obj);

/**
 * Send the next byte
 *
 * @param env JNI env that can be used by the method
 */
void sendNextByte(JNIEnv* env, jobject obj);
```

In Listing 6, all of the configuration data for the serial port is stored in the Java object returned by the `initializeSerialPort()` method, and Java code is in full control of setting each individual bit in the hardware. Several issues with the version in Listing 6 will lead to poorer performance than the version in Listing 7:

Listing 7. Interface to a serial port: Version 2

```
/**
 * Initializes the serial port and returns an opaque handle to a native
 * structure that contains the hardware address for the serial port
 * and holds information needed by the serial port such as
 * the next buffer to write data into
 *
 * @param env JNI env that can be used by the method
 * @param comPortName the name of the serial port
 * @returns opaque handle to be passed to setSerialPortByte and
 *          getSerialPortByte calls
 */
jlong initializeSerialPort2(JNIEnv* env, jobject obj, jstring comPortName);

/**
 * sends a byte on the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig opaque handle for the serial port
 * @param byte the byte to be sent
 */
void sendSerialPortByte(JNIEnv* env, jobject obj, jlong serialPortConfig,
    jbyte byte);

/**
 * Reads the next byte from the serial port
 *
 * @param env JNI env that can be used by the method
 * @param serialPortConfig opaque handle for the serial port
 * @returns the byte read from the serial port
 */
jbyte readSerialPortByte(JNIEnv* env, jobject obj,  jlong serialPortConfig);
```

Performance Tip #6

Structure the application's data so that it exists on the right side of the boundary and can be accessed by the code that uses it without requiring many transitions across the Java/native boundary.

The most obvious issue is that the interface in Listing 6 requires a JNI call for each bit set or retrieved, as well as a JNI call to read a byte from, or write a byte to, the serial port. This leads to nine times as many JNI calls for each byte read or written. The second issue is that LListing 6 stores the configuration information for the serial port in a Java object that's on the wrong side of the Java/native boundary for where the data is used. We need this configuration data only on the native side; storing it on the Java side will cause numerous callbacks from the native to Java to set/get this configuration information. Listing 7 stores the configuration information in a native structure (for example, a C `struct`) and returns an opaque handle to Java code, which can be returned on subsequent calls. This means that when a native is running it can reach directly into the structure without needing to call back to Java code for information such as the serial-port hardware address or the next buffer available. The performance of an implementation using Listing 7 will therefore be much better.

Using many local references without informing the JVM

Local references are created for any object returned by a JNI function. For example, when you call `GetObjectArrayElement()`, a local reference to the object in the array is returned. Consider how many local references are used when the code in Listing 8 is run on a very large array:

Listing 8. Creating local references

```
void workOnArray(JNIEnv* env, jobject obj, jarray array){
   jint i;
   jint count = (*env)->GetArrayLength(env, array);
   for (i=0; i < count; i++) {
      jobject element = (*env)->GetObjectArrayElement(env, array, i);
      if((*env)->ExceptionOccurred(env)) {
         break;
      }

      /* do something with array element */
   }
}
```

Each time `GetObjectArrayElement()` is called, a local reference is created for the element and isn't freed until the native completes. The larger the array, the more local references will be created.

Performance Tip #7

When a native causes the creation of a large number of local references, delete each reference when it is no longer required.

These local references are freed automatically when the native method terminates. The JNI specification requires that each native be able to create at least 16 local references. Although this is adequate for many methods, some methods need to access more during their lifetime. In this case, you should either delete references that are no longer needed, by using the JNI `DeleteLocalRef()` call, or inform the JVM that you'll be using a larger number of local references.

Listing 9 adds a call to `DeleteLocalRef()` to the example in Listing 8, informing the JVM that the local reference is no longer needed and limiting the number of local references that exist at one time to a reasonable number, regardless of the array's size:

Listing 9. Adding `DeleteLocalRef()`

```
void workOnArray(JNIEnv* env, jobject obj, jarray array){
   jint i;
   jint count = (*env)->GetArrayLength(env, array);
   for (i=0; i < count; i++) {
      jobject element = (*env)->GetObjectArrayElement(env, array, i);
      if((*env)->ExceptionOccurred(env)) {
         break;
      }

      /* do something with array element */

      (*env)->DeleteLocalRef(env, element);
   }
}
```

Performance Tip #8

If a native will have a large number of local references simultaneously, call the JNI `EnsureLocalCapacity()` method to inform the JVM and allow it to optimize its handling of local references for this case.

You can call the JNI `EnsureLocalCapacity()` method to tell the JVM that you'll be using more than 16 local references. This allows the JVM to optimize the handling of local references for

that native. Failure to inform the JVM can lead to a `FatalError` if the required local references cannot be created, or poor performance that's due to a mismatch between the local-reference management employed by the JVM and the number of local references used.

Correctness pitfalls

The top five JNI correctness pitfalls are:

- Using the wrong `JNIEnv`
- Not checking for exceptions
- Not checking return values
- Using array methods incorrectly
- Using global references incorrectly

Using the wrong `JNIEnv`

A thread executing native code uses a `JNIEnv` to make JNI methods calls. But the `JNIEnv` is used for more than just dispatching the requested methods. The JNI specification states that each `JNIEnv` is local to a thread. A JVM can rely on this assumption, storing additional thread-local information within the `JNIEnv`. Use of the `JNIEnv` from one thread by another thread can lead to subtle bugs and crashes that are hard to debug.

> **Correctness Tip #1**
>
> Only use the `JNIEnv` with the single thread to which it is associated.

A thread can get a `JNIEnv` by calling `GetEnv()` using the JNI invocation interface through a `JavaVM` object. The `JavaVM` object itself can be obtained by calling the JNI `GetJavaVM()` method using a `JNIEnv` object and can be cached and shared across threads. Caching a copy of the `JavaVM` object enables any thread with access to the cached object to get access to its own `JNIEnv` when necessary. For optimal performance, however, a thread should pass the `JNIEnv` that it received when it was invoked down through the methods it calls, because looking it up can require significant work.

Not checking for exceptions

Many of the JNI methods that natives can call can raise exceptions on the executing thread. When Java code executes, these exceptions cause a change to the execution flow such that the exception-handling code path is automatically invoked. When a native makes a call to a JNI method, an exception can be raised, but it's up to the native to check for exceptions and take appropriate action. A common JNI programming pitfall is to call a JNI method and to proceed without checking for exceptions once the call is complete. This can lead to buggy code and crashes.

For example, consider code that calls `GetFieldID()`, which raises the `NoSuchFieldError` if the requested field can't be found. If the native code proceeds without checking for the exception and uses the field ID it thought was returned, a crash can occur. The code in Listing 10, for example, might cause a crash — rather than throw a `NoSuchFieldError`— if the Java class is modified so that the `charField` field no longer exists:

Listing 10. Failing to check for exceptions

```
jclass objectClass;
jfieldID fieldID;
jchar result = 0;

objectClass = (*env)->GetObjectClass(env, obj);
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
result = (*env)->GetCharField(env, obj, fieldID);
```

Correctness Tip #2

Always check for exceptions after making JNI calls that can raise exceptions.

It's much easier to include the code to check for the exception than to try to debug a crash later on. Often you can simply check if an exception has occurred and if so return immediately to Java code so that the exception is thrown. It will then be either handled or displayed using the normal Java exception-handling process. For example, Listing 11 checks for exceptions:

Listing 11. Checking for exceptions

```
jclass objectClass;
jfieldID fieldID;
jchar result = 0;

objectClass = (*env)->GetObjectClass(env, obj);
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
if((*env)->ExceptionOccurred(env)) {
   return;
}
result = (*env)->GetCharField(env, obj, fieldID);
```

Not checking and clearing exceptions can lead to unexpected behavior. Can you spot what is wrong with this code?

```
fieldID = (*env)->GetFieldID(env, objectClass, "charField", "C");
if (fieldID == NULL){
   fieldID = (*env)->GetFieldID(env, objectClass,"charField", "D");
}
return (*env)->GetIntField(env, obj, fieldID);
```

The problem is that even though the code handles the case in which the initial `GetFieldID()` doesn't return the field ID, it does not *clear* the exception that this call would set. The return from the native will therefore cause an exception to be thrown immediately.

Not checking return values

Many JNI methods have a return value that indicates whether the call succeeded or not. A common pitfall, similar to not checking for exceptions, is to fail to check the return value and for the code to proceed on the assumption that the call was successful. For most of the JNI methods, the return value and exception status will both be set so that checking either the exception status or the return value will let the application know if the method ran correctly or not.

Correctness Tip #3

Always check the return value from a JNI method and include code paths to handle errors.

Can you spot what is wrong with the following code?

```
clazz = (*env)->FindClass(env, "com/ibm/j9//HelloWorld");
method = (*env)->GetStaticMethodID(env, clazz, "main",
                    "([Ljava/lang/String;)V");
(*env)->CallStaticVoidMethod(env, clazz, method, NULL);
```

The problems are that if the `HelloWorld` class is not found or if the `main()` method doesn't exist, the native will cause a crash.

Using array methods incorrectly

The `Get`*`XXX`*`ArrayElements()` and `Release`*`XXX`*`ArrayElements()` methods allow you to request array elements. Similarly, `GetPrimitiveArrayCritical()`, `ReleasePrimitiveArrayCritical()`, `GetStringCritical()`, and `ReleaseStringCritical()` allow you to request array elements or string bytes to maximize the likelihood that they will get a direct pointer to the array or string. Two common pitfalls are associated with the use of these methods. The first is to forget to commit changes in the call to the `Release`*`XXX`*`()` method. There's no guarantee that you will actually get a direct pointer to the array or string even when using the `Critical` versions. Some JVMs will always return a copy, and in these JVMs the changes made to the array will not be copied back if you specify `JNI_ABORT` in the call to `Release`*`XXX`*`()` or forget to call `Release`*`XXX`*`()`.

For example, consider this code:

```
void modifyArrayWithoutRelease(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)-> (*env)->GetByteArrayElements(env,arr1,&isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   buffer[0] = 1;
}
```

Correctness Tip #4

Don't forget to call `Release`*`XXX`*`()` with a mode of `0` (copy back and free the memory) for each `Get`*`XXX`*`()` call.

On a JVM that provides a direct pointer to the array, the array will be updated; however, on a JVM that returns a copy it will not be. This can lead to cases where your code seems to work on some JVMs but fails to work on others. You should always include a release call, as shown in Listing 12:

Listing 12. Including a release call

```
void modifyArrayWithRelease(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)-> (*env)->GetByteArrayElements(env,arr1,&isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   buffer[0] = 1;

   (*env)->ReleaseByteArrayElements(env, arr1, buffer, JNI_COMMIT);
   if ((*env)->ExceptionCheck(env)) return;
}
```

The second pitfall is not honoring the restrictions placed by the specification on code that executes between the `Get`*`XXX`*`Critical()` and `Release`*`XXX`*`Critical()`. The native may not make any JNI calls between these methods and may not block for any reason. Failing to honor these restrictions can lead to intermittent deadlock within the application or the JVM as a whole.

For example, the following code might look okay:

```
void workOnPrimitiveArray(JNIEnv* env, jobject obj, jarray arr1) {
   jboolean isCopy;
   jbyte* buffer = (*env)->GetPrimitiveArrayCritical(env, arr1, &isCopy);
   if ((*env)->ExceptionCheck(env)) return;

   processBufferHelper(buffer);

   (*env)->ReleasePrimitiveArrayCritical(env, arr1, buffer, 0);
   if ((*env)->ExceptionCheck(env)) return;
}
```

Correctness Tip #5

Ensure the code does not make any JNI calls or block for any reason between calls to `Get`*`XXX`*`Critical()` and `Release`*`XXX`*`Critical()`.

However, we need to validate that all of the code that can be run when `processBufferHelper()` is called does not violate any of the restrictions. These restrictions apply to all code that executes between the `Get` and `Release` calls, whether it is part of the native itself or not.

Using global references incorrectly

Natives can create global references so that objects are not garbage collected until they are no longer needed. Common pitfalls are forgetting to delete global references that have been created or losing track of them completely. Consider a native that creates a global reference but does not delete or store it anywhere:

```
lostGlobalRef(JNIEnv* env, jobject obj, jobject keepObj) {
   jobject gref = (*env)->NewGlobalRef(env, keepObj);
}
```

Correctness Tip #6

Always keep track of global references and ensure they are deleted when the object is no longer required.

When the global reference is created, the JVM adds it to a list that excludes that object from garbage collection. When the native returns, not only has it not freed the global reference, but the application also no longer has a way to get the reference in order to free it later — so the object will live forever. Not freeing global references causes issues not only because they keep the object itself alive but also because they keep alive all objects that can be reached through the object. In some cases this can add up to a significant memory leak.

Avoiding the common pitfalls

Suppose you have just finished writing some new JNI code or inherited some JNI code from elsewhere. How can you ensure that you have avoided the common pitfalls or can find them in

your inherited code? Table 1 identifies the techniques you can use to root out instances of the common pitfalls:

Table 1. Checklist for identifying JNI programming pitfalls

	Not caching	Triggering array copies	Wrong boundary	Reaching back too much	Using many local references	Using wrong JNIEnv	Not checking for exceptions	Not checking for return values	Using arrays incorrectly	Using global references incorrectly
Validation against specification						X	X		X	
Method tracing	X	X	X	X			X		X	X
Dumps										X
`-verbose:jni`					X					
Code review	X	X	X	X	X	X	X	X	X	X

You can identify many of the common pitfalls early in the development cycle by:

- Validating new code against the specification
- Analyzing the method trace
- Using the `-verbose:jni` option
- Generating dumps
- Performing code reviews

Validating new code against the JNI specification

It's a good practice to maintain a list of the constraints imposed by the specification and review natives for compliance with the list, either manually or through automatic code analysis. You'll likely expend much less effort ensuring compliance than you would debugging the subtle and intermittent failures that can occur when the constraints are not observed. Here's a starting list of specification-conformance checks to do for newly developed code (or code that is new to you):

- Validate that a `JNIEnv` is used only with the thread to which it is associated.
- Validate that JNI Methods are not called within `Get`*`XXX`*`Critical()`'s `Release`*`XXX`*`Critical()` section.
- For a method that enters a critical section, validate that the method does not return before it is released.
- Validate that there is a check for an exception after all JNI calls that can raise an exception.
- Ensure that all `Get`/`Release` calls are matched within each JNI method.

IBM's JVM implementation includes an option that turns on automatic JNI checks, at the cost of slower execution. In conjunction with good unit tests for your code, this is a powerful tool. You can run the application or unit tests once to do a compliance check or when you encounter bugs for which you suspect natives to be the cause. In addition to doing the specification-compliance checks we listed above, it also ensures that:

- The parameters passed to JNI methods are of the correct types.
- JNI code does not read past the ends of arrays.
- Only valid pointers are passed to JNI methods.

Not all the findings the JNI check reports are necessarily errors in the code. They include suggestions as to code that should be reviewed closely to ensure that it does what was intended.

You enable the JNI check option with the following command line:

```
Usage: -Xcheck:jni:[option[,option[,...]]]

        all             check application and system classes
        verbose         trace certain JNI functions and activities
        trace           trace all JNI functions
        nobounds        do not perform bounds checking on strings and arrays
        nonfatal        do not exit when errors are detected
        nowarn          do not display warnings
        noadvice        do not display advice
        novalist        do not check for va_list reuse
        valist          check for va_list reuse
        pedantic        perform more thorough, but slower checks
        help            print this screen
```

Using the IBM JVM's `-Xcheck:jni` option as part of the standard development process can help you find coding errors much more easily. In particular, it can help you root out the pitfalls of using the `JNIEnv` with the wrong thread and using critical regions incorrectly.

Recent Sun JVMs provide a similarly named `-Xcheck:jni` option. It operates differently from the IBM version and provides different information, but the purpose is the same. It issues warnings when it sees code that doesn't conform to the specification and can help you find instances of the common JNI pitfalls.

Analyzing the method trace

Generating a trace of the natives that are invoked, along with the JNI callbacks that these natives make, can be useful for rooting out a number of the common pitfalls. Issues to look for include:

- An abundance of `GetFieldID()` and `GetMethodID()` calls — in particular, if the calls are for the same fields and methods — indicates that the fields and method are not being cached.
- Instances of calls to `Get`*`Type`*`ArrayElements()` instead of `Get`*`Type`*`ArrayRegion()` can indicate unnecessary copying.
- Switching quickly back and forth between Java code and natives (as indicated by timestamps) can indicate the wrong boundary between Java code and natives, leading to poor performance.
- The pattern in which each invocation of a native function being followed by a number of `GetFieldID()` calls can indicate that instead of passing the parameters required, you are forcing the natives to reach back for the data needed to complete their work.
- The lack of calls to `ExceptionOccurred()` or `ExceptionCheck()` after calls to JNI methods that can throw exceptions can indicate that the natives are not properly checking for exceptions.
- A mismatch between the number of `Get`*`XXX`*`()` and `Release`*`XXX`*`()` method calls can indicate missing releases.

- Calls to JNI methods between `Get`*`XXX`*`Critical()` and `Release`*`XXX`*`Critical()` calls indicate that the constraints imposed by the specification are not being observed.
- If the elapsed time between the calls to `Get`*`XXX`*`Critical()` and `Release`*`XXX`*`Critical()` is long, this can indicate that the constraint imposed by the specification not to make blocking calls is not being observed.
- A large imbalance between calls to `NewGlobalRef()` and `DeleteGlobalRef()` can indicate a failure to free global references when they are no longer needed.

Some JVM implementations provide a mechanism through which a method trace can be generated. You can also generate a trace through external tools such as profilers and code-coverage tools.

The IBM JVM implementation provides a number of ways to generate trace information. The first is to use the `-Xcheck:jni:trace` option. This generates a trace of the native methods called as well as the JNI callbacks that they make. Listing 13 shows an excerpt of a trace (with some lines split for readability only):

Listing 13. Method trace generated by the IBM JVM implementation

```
Call JNI: java/lang/System.getPropertyList()[Ljava/lang/String; {
00177E00   Arguments: void
00177E00   FindClass("java/lang/String")
00177E00   FindClass("com/ibm/oti/util/Util")
00177E00   Call JNI: com/ibm/oti/vm/VM.useNativesImpl()Z {
00177E00     Arguments: void
00177E00     Return: (jboolean)false
00177E00   }
00177E00   Call JNI: java/security/AccessController.initializeInternal()V {
00177E00     Arguments: void
00177E00     FindClass("java/security/AccessController")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedAction;)Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedExceptionAction;)Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedAction;Ljava/security/AccessControlContext;)
             Ljava/lang/Object;")
00177E00     GetStaticMethodID(java/security/AccessController, "doPrivileged",
             "(Ljava/security/PrivilegedExceptionAction;
             Ljava/security/AccessControlContext;)Ljava/lang/Object;")
00177E00     Return: void
00177E00   }
00177E00   GetStaticMethodID(com/ibm/oti/util/Util, "toString",
             "([BII)Ljava/lang/String;")
00177E00   NewByteArray((jsize)256)
00177E00   NewObjectArray((jsize)118, java/lang/String, (jobject)NULL)
00177E00   SetByteArrayRegion([B@0018F7D0, (jsize)0, (jsize)30, (void*)7FF2E1D4)
00177E00   CallStaticObjectMethod/CallStaticObjectMethodV(com/ibm/oti/util/Util,
             toString([BII)Ljava/lang/String;, (va_list)0007D758) {
00177E00     Arguments: (jobject)0x0018F7D0, (jint)0, (jint)30
00177E00     Return: (jobject)0x0018F7C8
00177E00   }
00177E00   ExceptionCheck()
```

The trace excerpt in Listing 13 shows the native being called (for example, `AccessController.initializeInternal()V`) and then the JNI callbacks the native makes.

Using the `-verbose:jni` option

Both the Sun and IBM JVMs also provide a `-verbose:jni` option. For the IBM JVM, turning this option on provides information about what JNI callbacks are being made. Listing 14 shows an example:

Listing 14. Listing JNI callbacks with the IBM JVM's `-verbose:jni`

```
<JNI GetStringCritical: buffer=0x100BD010>
<JNI ReleaseStringCritical: buffer=100BD010>
<JNI GetStringChars: buffer=0x03019C88>
<JNI ReleaseStringChars: buffer=03019C88>
<JNI FindClass: java/lang/String>
<JNI FindClass: java/io/WinNTFileSystem>
<JNI GetMethodID: java/io/WinNTFileSystem.<init> ()V>
<JNI GetStaticMethodID: com/ibm/j9/offload/tests/HelloWorld.main ([Ljava/lang/String;)V>
<JNI GetMethodID: java/lang/reflect/Method.getModifiers ()I>
<JNI FindClass: java/lang/String>
```

For the Sun JVM, turning the `-verbose:jni` option on doesn't provide information about the calls being made, but it does provide additional information about the natives used. Listing 15 shows an example:

Listing 15. Using the Sun JVM's `-verbose:jni`

```
[Dynamic-linking native method java.util.zip.ZipFile.getMethod ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.initIDs ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.init ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.inflateBytes ... JNI]
[Dynamic-linking native method java.util.zip.ZipFile.read ... JNI]
[Dynamic-linking native method java.lang.Package.getSystemPackage0 ... JNI]
[Dynamic-linking native method java.util.zip.Inflater.reset ... JNI]
```

Turning on this option also causes the JVM to emit warnings when too many local references are used without informing the JVM. For example, the IBM JVM generates messages like this one:

```
JVMJNCK065W JNI warning in FindClass: Automatically grew local reference frame capacity
from 16 to 48. 17 references are in use.
Use EnsureLocalCapacity or PushLocalFrame to explicitly grow the frame.
```

Although the `-verbose:jni` and `-Xcheck:jni:trace` options make it easy to get the information you need, reviewing this information manually can take a fair amount of effort. It's a good idea to create scripts or utilities that can process the trace files being generated by your JVM and look for the warning signs.

Generating dumps

Dumps generated from a running Java process contain a wealth of information about a JVM's state. For many JVMs, they include information about global references. For example, the recent Sun JVMs include the following line in the dump information:

```
JNI global references: 73
```

By generating before and after dumps, you can assess if you're creating any global references that are not being freed when they should be.

You can request a dump in UNIX® environments by issuing a `kill -3` or `kill -QUIT` on the `java` process. On Windows®, use Ctrl+Break.

For the IBM JVM, use these steps to obtain information on global references:

1. Add `-Xdump:system:events=user` to the command line. This asks the JVM to generate a system dump when you invoke `kill -3` on a UNIX variant or Ctrl+Break on Windows.
2. When your program is running, generate subsequent dumps.
3. Run `jextract -nozip core.XXX output.xml`, which extracts dump information into a readable format in output.xml.
4. Look for `JNIGlobalReference` entries in output.xml, which give you information about the current global references, as shown in Listing 16:

Listing 16. `JNIGlobalReference` entries in output.xml

```
<rootobject type="Thread" id="0x10089990" reachability="strong" />
<rootobject type="Thread" id="0x10089fd0" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x100100c0" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011250" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011840" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10011880" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10010af8" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10010360" reachability="strong" />
<rootobject type="JNIGlobalReference" id="0x10081f48" reachability="strong" />
<rootobject type="StringTable" id="0x10010be0" reachability="weak" />
<rootobject type="StringTable" id="0x10010c70" reachability="weak" />
<rootobject type="StringTable" id="0x10010d00" reachability="weak" />
<rootobject type="StringTable" id="0x10011018" reachability="weak" />
```

By looking at the numbers reported in subsequent Java dumps, you can assess if global references are being leaked.

See Resources for additional information on using the dump files and `jextract` with the IBM JVM.

Performing code reviews

Code reviews can often be effective for spotting common pitfalls and can be done at a number of levels. When you inherit new code, a quick scan can reveal issues that would take much longer to debug later on. In some cases, a review is the only way to identify an instance of a pitfall such as the code not checking return values. For example, the problem with this code would likely be easy to identify through a code review but much harder to find through debugging:

```
int calledALot(JNIEnv* env, jobject obj, jobject allValues){
   jclass cls = (*env)->GetObjectClass(env,allValues);
   jfieldID a = (*env)->GetFieldID(env, cls, "a", "I");
   jfieldID b = (*env)->GetFieldID(env, cls, "b", "I");
   jfieldID c = (*env)->GetFieldID(env, cls, "c", "I");
   jfieldID d = (*env)->GetFieldID(env, cls, "d", "I");
   jfieldID e = (*env)->GetFieldID(env, cls, "e", "I");
   jfieldID f = (*env)->GetFieldID(env, cls, "f", "I");

}

jclass getObjectClassHelper(jobject object){
   /* use globally cached JNIEnv */
   return cls = (*globalEnvStatic)->GetObjectClass(globalEnvStatic,allValues);
}
```

A code review would likely identify that the first method is not properly caching field IDs even though the same IDs are used repeatedly, and that the second method is using a `JNIEnv` on threads other than the one the `JNIEnv` should be used on.

Conclusion

You're now aware of the top 10 JNI programming pitfalls and have learned some good practices for identifying them in existing or new code. Apply these practices diligently to increase the likelihood that your JNI code is correct and that your application can achieve the performance levels it requires.

The ability to integrate existing code assets efficiently is essential to succeeding with two technologies that are gaining momentum: service-oriented architecture (SOA) and cloud-based computing. JNI is a key technology for integrating non-Java legacy code and components into a Java-based platform used as a building block for an SOA or cloud-based system. Proper use of JNI can speed the process of service-enabling these components and allow you to derive the maximum advantage from existing investments.

Resources

Learn

- See IBM Bluemix in action: In this demo, David Barnes shows you how to develop, create, and deploy an application in the cloud.
- Java Native Interface: You'll find the JNI specification, FAQ, examples, and other resources here.
- "Java programming with JNI" (Scott Stricker, developerWorks, March 2002): In this tutorial, learn about JNI essentials and some more-advanced programming challenges.
- IBM Java SDK Info Center: Learn more about using the dump files and `jextract` with the IBM JVM.
- "JNI Programming on AIX" (Nikolay Yevik, developerWorks, March 2004): Get general guidance for developing JNI applications using the IBM JDK for AIX.
- "Design and Implementation of a Comprehensive Real-time Java Virtual Machine" (Joshua Auerbach et al., *Proceedings of the Seventh ACM and IEEE International Conference on Embedded Software*, 2007): Read about a JVM that returns array copies.
- Browse the technology bookstore for books on these and other technical topics.
- developerWorks Java technology zone: Find hundreds of articles about every aspect of Java programming.

Get products and technologies

- Download IBM product evaluation versions or explore the online trials in the IBM SOA Sandbox and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

- Get involved in the My developerWorks community.

About the authors

Michael Dawson

Michael Dawson graduated in 1989 from the University of Waterloo with a bachelor's degree in computer engineering and in 1991 from Queens University with a master's degree in electrical engineering, specializing in cryptography. He then spent a number of years doing security consulting and developing products at companies ranging from start-ups to IBM. He has held leadership roles in teams developing e-commerce applications and delivering them as services including EDI communication services, credit card processing, online auctions, electronic invoicing, and JVMs. The technologies used ranged from C/C++ to Java and Java EE platforms and components across a range of operating systems. Michael joined IBM in 2006. He works on the J9 JVM team implementing Java class libraries and core JVM components.

Graeme Johnson

Graeme Johnson is a development manager and technical lead on IBM's J9 Virtual Machine team. He has been developing virtual machines and debuggers since he joined IBM (previously Object Technology International) in 1994, and has worked on both VisualAge for Java and IBM/OTI Smalltalk runtimes. Recently Graeme has been focusing on the Apache Harmony project, and the Java/PHP runtime support for IBM's Project Zero. Graeme is a regular conference speaker on a variety of topics including: Apache Harmony at JavaOne 2006, multiplatform C development at EclipseCon 2007, and an examination of the PHP runtime at International PHP 2006.

Andrew Low

Andrew Low joined Object Technology International Inc. (OTI) in 1994 as a university graduate after working as a co-op student for several terms. He remained with the company when IBM acquired OTI in 1996. His work has always been associated with VM technology, ranging from the very small (cell phones and PDAs) to the very large (IBM zSeries mainframe systems). He is currently a technical leader on the J9 VM team at the IBM Ottawa Lab, helping shape the core technology behind IBM's Java runtimes. Andrew has played a key role in the development of the J9 VM and is a recognized expert in embedded systems and the Java ME marketplace. Recently he has been involved in several strategic efforts to push Java runtime technology into the Web2.0 world.