Java Native Methods

Alligator Descartes

http://www.symbolstone.org

November 16, 2000

"Java Native Methods" was written, illustrated and prepared by Alligator Descartes using the DocBook DTD, OpenJade and LaTeX.

Notes on this edition

This book is the virtually complete text of a book called "Java Native Methods". Note that this material has not undergone final editing and that Chapter 14, "Native Rendering from Java" is not completely finished. Furthermore, there is no index for this book.

These points notwithstanding, this is a (virtually) complete book discussing native method programming with both the JNI (Java Native Interface) and RNI (Raw Native Interface) APIs.

Demonstration Code

The full demonstration code for this book can be downloaded from:

```
http://www.symbolstone.org/technology/java/nmbook/nmbook_src.tar.gz
http://www.symbolstone.org/technology/java/nmbook/nmbook_src.zip
```

Due to some issues with the formatting engine used, some of the demonstration code examples within the pages of the book have suffered line-wrap and hyphenation where it should not have occurred. The downloadable demonstration code does not suffer from this problem.

Author

Alligator Descartes has been an itinerant fiddler with computers from a very early age ruined only by obtaining a BSc in Computer Science from the University of Strathclyde, Glasgow. His computing credits include several years of Oracle DBA work, multi-user Virtual Reality servers, high-performance 3D graphics programming and several Perl modules. His spare time is spent trudging around Scotland looking for stone circles and Pictish symbol stones to photograph.

Alligator Descartes is not his real name.

Table of Contents

List of Tables

List of Figures

List of Examples

Chapter 1. Preface

Java is arguably the most popular cross-platform programming language in use today, with ports available for almost any operating system, from mainframes down to handheld PCs. However, with the panacea of platform-independence comes the price of potentially reduced performance and the inability to directly access hardware-dependent features, a major problem in the field of 3D graphics and other hardware-accelerated areas.

To address these issues and provide the developer with a way to access this important functionality, Java provides a mechanism called *native methods,* an technique that allows your code to seamlessly switch execution from Java into compiled, platform-specific code. This book discusses the two most common APIs in use today, the widely used Java Native Interface (JNI) and the Microsoft-specific Raw Native Interface (RNI).

I have tried to provide a logical path through the various aspects of both native method interfaces such as basic techniques to manipulate variables, strings and arrays, to more complex techniques such as interfacing with the garbage collector and management of references. When possible, both the JNI and RNI are discussed together as many of the concepts behind a topic are identical for both interfaces, differing only in the actual API calls.

However, in some cases, the differences in the way that the JNI and RNI address problems are so radically different that I have dedicated a chapter entirely to each interface.

Here's a rundown of the book, chapter by chapter:

Chapter 2, *Introduction to Java Native Methods*

This chapter introduces the topic of native methods and some good reasons to use them. It also discusses when you should consider using them and when it might not be worth using them.

I also introduce the examples used throughout the book.

Chapter 3, *Java Architecture*

This chapter discusses the basic architecture used by the Java Virtual Machine and how native methods fits into that overall design. We also discuss the basic concepts of class files and how data types are used by the native interfaces. This information is used again and again throughout the book, so this chapter is recommended reading.

We also discuss the concrete reality of trying to write native method code that will work on a variety of platforms. I outline the principles behind, and examples of, compiling and linking dynamically loadable libraries on UNIX, Windows, MacOS and OS/2 platforms.

Chapter 4, *Introducing the Java Native Interface*

This chapter discusses the architecture behind the JNI and how it interacts with the Java Virtual Machine. We also discuss how data type mapping occurs within the JNI and what the JNI-specific data types are.

A simple example is discussed showing how you call a native method from Java and how to implement your first native method.

Chapter 5, *Introducing the Raw Native Interface*

This chapter discusses the architecture behind the RNI and how it interacts with the Microsoft Java Virtual Machine. Also discussed is how the standard C data types map to the data types used by the RNI.

A simple example is discussed showing how you call a native method from Java and how to implement your first native method.

Chapter 6, *Working with Strings*

This chapter dissects the way in which Java strings are created and manipulated using both the JNI and the RNI.

Chapter 7, *Manipulating Arrays*

This chapter discusses how arrays can be created and manipulated from both the JNI and the RNI.

Chapter 8, *Class Operations with the JNI*

This chapter is dedicated to the JNI and discusses the way that information stored within a class can be retrieved, for example, values stored within variables. Secondly, we discuss how we can invoke methods at both a class-level and instance-level. Thirdly, we look into the techniques behind creating and manipulating objects with the JNI and, finally, we discuss how meta-information about a class can be retrieved and used within your code.

This chapter is JNI-specific.

Chapter 9, *Class Operations with the RNI*

This chapter contains the same content as Chapter 8, but only discusses the RNI.

Chapter 10, *Exception Handling*

Java typically uses an exception-based error-triggering and handling system as opposed to a pure return code system used by languages such as C.

This chapter discusses how you can trigger exceptions from within native code, or safely handle exceptions thrown when your native code is executing.

Chapter 11, *Threading*

This chapter discusses how to manage multi-threading in native code.

Chapter 12, *Memory Management*

This chapter covers one of the more complex areas of native method programming, that of interacting with the Virtual Machine's garbage collector. I explain the ins and outs of what the garbage collector is doing, and how this can be problematic with writing native method code. We also discuss how our native method code can explicitly control the garbage collector for performance reasons.

Chapter 13, *Embedding a JVM*

This chapter discusses one of the more exciting uses of native method programming which allows you to embed a Java Virtual Machine into an executable program. This is almost the complete opposite technique to the way that the book has been structured so far in that we will now be talking about a compiled program making Java calls, rather than a Java program making native calls!

We'll discuss how to embed a JVM into programs using both the RNI and the JNI and also discuss how to register native methods with the JNI in a way that gives you better application security.

Chapter 14, *Native Rendering from Java*

This chapter looks at some of the issues associated with rendering graphics from native code onto Java AWT and Swing drawing surfaces.

Resources

As further reference to the material in this book, there are a few books and web sites that discuss the topic of native method programming:

http://www.javasoft.com

JavaSoft's home page. This site contains the downloadable specification for the Java Native Interface and also contains a very simple tutorial on native method programming.

You can also download the Java Development Kit from this site.

http://www.microsoft.com/java

Microsoft's Java Development home page. This site contains the specification for the Raw Native Interface. There is very little in the way of actual example code or tutorials.

You can download the Microsoft Java Software Development Kit from here.

Essential JNI, by Rob Gordon

"Essential JNI" covers the JNI API (including Java-2) extensively, but, as the title would suggest, does not cover RNI.

The Java Native Interface: Programmer's Guide and Specification, by Sheng Liang

This book, written by the designer of JNI, is a comprehensive look at the interface.

Concurrent Programming in Java, by Doug Lea

This excellent book covers the topic of multi-threading within the Java environment and threading design patterns and methodologies.

Acknowledgements

I would like to thank, first and foremost, my wife Carolyn for once again having to watch the painful process of me writing yet another book.

Martin McCarthy deserves a magnum of Trappist beer (with accompanying packet of Alka Selzter) for proofreading drafts of the book and trying out the example code. Chet Haase also managed to survive reading early drafts and review copies and is no doubt a better person for the experience! And, last, but not least, Rob "Telescope Boy" DeMillo seems also to have survived the process with a modicum of self-respect.

Thanks, and mountain survival gear, go to Paula Ferguson for doing a fabulous editing job and sticking with the book through the long period which it took to write and bang into shape.

Finally, thanks to the people that used, and complained about, the Magician Java/OpenGL bindings over the years. The experience of writing that software enriched the information in this book no end. In no particular order: Jason "Mr. Wiggles" Osgood, Hayden Schultz, Jon Leech, Bruce D'Amora, Tim Donovan, Jack Middleton, David Yu, Michael Gold and Suzy Deffeyes.

Chapter 2. Introduction

The Java programming language developed by Sun Microsystems has captured the attention of the computing industry since it was introduced in 1995. The language boasts various features that are attractive to computing professionals, such as platform-independence, reusable object-oriented modules, and the heady claim of “Write once, run anywhere”. Java is not without its drawbacks, however. Two major concerns involve legacy code and the performance of Java programs.

Java source code is compiled into platform-independent byte codes that are designed to execute within a Java Virtual Machine (JVM). The compiled byte codes are then loaded into the JVM for execution. The problem is that the JVM adds a layer of translation between the software and the processor of the computer that it is running on, which can cause a fairly dramatic slowdown in performance. This is the first “popularly acclaimed” major problem with Java.

The second, more pervasive, problem concerns legacy code. With the advent of such a popular computing environment, many developers have abandoned other languages and jumped on the Java bandwagon. However, completely rewriting an application to take full advantage of a Java-centric system involves sweeping aside all the code written in other languages. If this is done on a large scale, thousands, if not millions, of programmer-hours of software development may be lost.

Native methods offer a solution to both of these problems. A *native method* is essentially a function declared within Java source code that is actually implemented using “native” code—code written in another language and compiled for a specific processor. Since native methods are compiled for a particular platform, you can use them to bypass the performance limitations inherent in the Java environment. In addition, with some design work, native methods can be used to integrate legacy software into Java-centric systems.

Unfortunately, a lot of information regarding native method programming is hearsay at best and disinformation at worst. In the rest of this chapter, I plan to set the record straight about native methods—both dispelling some myths about them and making it clear what you are getting into when you decide to use them. My goal is to help you understand when it is appropriate to use native methods and when it is best to avoid them completely.

Programming Complexity

Without a doubt, native methods are extremely tricky to program. This is one of the major issues that you must take into consideration when you are deciding whether or not a particular application should use native methods.

Programming native methods might appear to be a straightforward task, a case of simply following the interfaces defined by the JVM you are targetting and then writing the code. What you must realize is that with native methods you are interacting directly with the JVM in many places, and this in itself is an art form. There are many occasions when particular operations can cause the JVM to crash. The JVM can also push a stick through your spokes and induce bizarre behavior within your native code. All this can make programming robust native methods extremely difficult.

Another issue that affects programming complexity is portability across operating systems. Once you decide to use native methods, you lose the inherent portability that Java promises. Now you have to decide which platforms to support. Should you support Windows 95/98/NT/2000, as well as Unix? Instead of Unix? Or just various Unix platforms? For each operating system you choose to support, you will probably have to make subtle modifications in order to compile and execute your code correctly on that platform. And if you want to use operating-system-specific functionality, such as integration with X/Motif or support for COM, you are looking at massive rewrites.

Performance

In the early days of Java Virtual Machine development, native methods offered a surefire way to improve the performance of an application. Using native methods for compute-intensive functionality could often cut the execution time for an application by at least a factor of three. And for particularly intensive tasks, speed increases of a factor of ten were not unheard of.

Today’s JVMs, however, are a far more sophisticated bunch. They not only have been internally optimized on various operating systems for maximum performance, but may also have been enhanced with “just-in-time compilation”. This feature, commonly referred to as JIT, involves compiling Java byte codes on-the-fly into native code that is then executed by the JVM. There is a small delay while the Java byte codes are compiled, but when a program is running, its speed of execution is extremely close, if not identical, to code written in C.

In addition, a native method incurs a slight additional overhead in execution time for *each invocation* of the method, due to the operation of “dropping into” native code from Java. For a short, frequently called operation, such as summing a small array of numbers, a native method may not prove to be particularly effective as a performance booster. In fact, short operations may take *longer* in native code than in Java! For long and computationally expensive operations, however, this delay occurs less often and is less significant in terms of overall performance.

What all this means is that native methods do not necessarily offer the speed-kick that was previously believed. If you are planning to use native methods for performance reasons, you should rigorously benchmark their effectiveness in your application before you commit to using them. For example, using native methods to provide high-performance 3D rendering or video decompression is probably a good idea, since these are extremely computationally intensive operations. However, using a native method to add two numbers together may not be a particularly good idea. If you benchmark some test operations, you should be able to get an idea of the performance increase you are likely to see with native methods.

When you perform your benchmark tests, you should run them on JVMs both with and without just-in-time compilation. That way you can determine whether the performance of your application will be acceptable on non-JIT JVMs. If the performance proves unacceptable, and you cannot guarantee that your users are going to be using JVMs with JIT, you may want to use native methods to even things out. In other words, even though native methods might not be necessary when your application is running in a JVM with JIT, they can make a big difference to users who don't have JIT.

Portability and Native Interfaces

As I already mentioned briefly, native methods are compiled for a specific platform, so they are inherently unportable. For example, native code written using a particular feature of Unix (e.g., the `select()` function), will not work on a Windows 95/98/NT/2000 system. To port the native method to the Windows platform, you need to replace the errant function with something Windows-specific. This may not be such a large problem with simple native methods that stick to using extremely standard functions and libraries, but in the case of native methods that interface with non-standard libraries such as X/Motif, OpenGL, or Direct X, your code will rapidly become extremely difficult to maintain. This is the curse of C- and C++-based software in industry today; entering into these realms forces you to abandon Java's cozy nest of platform-independence.

To make portability matters worse, each JVM can define and use its own native method interface (NMI). In this book, we'll discuss the two most prevalent ones: the newer Java Native Interface (JNI) supported by Java 1.1 and later, and the more low-level Raw Native Interface (RNI) that is based on the older NMI used in Java 1.0. You now have more branches of source code to support!

In other words, a Java program that uses native methods does not benefit from the "Write once, run anywhere" philosophy. In addition to the portability issues of supporting both Windows 95/98/NT/200 and Unix, you may have to worry about supporting various NMIs. This problem is particularly acute if you are using native methods in applets, where you have to support the two most popular browsers, Microsoft Internet Explorer and Netscape Communicator. In this scenario, one native method requires at least four separate code branches to support both platforms and both browsers.

Legacy Code

Within large Java projects, native methods are commonly used to provide a new unified front-end for legacy code. For example, in an organization that provides telephone sales, each agent may use a GUI-based form running under Windows to access a database. However, the database administration staff may well be using Unix workstations, and they might want to be able to test out the GUI for machine benchmarking or database performance. With a C-based application, this would require the GUI to be ported to a Unix windowing toolkit like Motif.

If instead the GUI is written in Java, both the sales agents and the administration team can use exactly the same program on different platforms. But now what do we do about all the complex legacy code that accesses and manipulates the database? Do we write off all that work and reimplement it in Java? Fortunately, with native methods, the answer is no. You can replace the GUI front-end with a Java-based solution for optimum portability, but encapsulate the legacy database connectivity code in native methods to take advantage of the work that has already been done.

Using native methods to integrate existing legacy code with new portable front-ends is a compelling solution to a widespread problem. As more and more companies develop intranets to provide access to corporate databases, they have to deal with the problems posed by heterogeneous systems. This is a scenario where it is easy to justify the use of Java native methods.

Applets or Applications?

Another thing to consider with regard to native methods is whether you are developing applets or full-blown Java applications. Applets give Java a sense of seamlessness within a networked environment. When a user loads a web page that has applets embedded within it, he can perform sophisticated tasks and interact directly with the content provided in the page. The ability to dynamically run programs without any direct user intervention is a powerful feature of Java. Unfortunately, native methods don’t mesh very well with the seamlessness of applets.

As the author of a Java applet, you don’t care which operating system or browser the user is using because Java is, after all, platform-independent. Your program is guaranteed to run safely on the user’s computer within the protective sandbox of the JVM. If your applet uses a native method, however, all of a sudden you do care about the user’s choice of operating system and browser. More importantly, in order for your native method to work, the user must install a code library on his system. This obviously requires user intervention, destroying the seamless nature of applets.

A related problem is that native code is inherently dangerous. A malicious or badly written native method could do untold damage to your hard disk or local area network. While Java is regarded as a secure language, native methods are not subject to any of Java’s security mechanisms. Native methods are inherently insecure, which may discourage users from downloading a native code library for use in an applet.

So, unless you are planning on providing a native interface that will prove hugely popular, you should not use native methods in applets that are available on the Internet. Native methods make it much more difficult to install and use an applet. Of course, if you are creating applets for a corporate intranet and you need native methods to interact with a legacy system, that’s a different matter. In this case, installation of the required software on each computer is simply part of the job of your information systems staff.

For Java applications, on the other hand, you don’t really have to worry about the installation process. A user, or system administrator, generally has to install an application on the target machine anyway, so there's no issue with having to install a native code library as well.

Speaking in Tongues

Until now, I haven't said anything about the language used to implement native methods. Java native methods can actually be implemented in either C or C++, since bindings exist for both languages.

So, which language should you use? The answer actually depends on a number of factors. If you are using native methods to integrate legacy code into a Java-based system, you clearly want to use the language of the legacy code, if that code is written in C or C++. (We'll talk in a minute about how to handle code that isn't written in one of these languages.) If you are going to be writing the native code while you are developing a Java application, you should use the language that you or your development team is most comfortable with.

All other things being equal, you simply have to choose. In my experience, with the RNI (or the very old Java 1.0 NMI), C is a far better choice—there are no benefits to be gained from using C++. However, if you are using the JNI, the C++ interface is far more intuitive. For example, with C++ you are far less likely to make silly mistakes with pointers. Consider the following C syntax:

```
JNIEnv *env;

(*env)->FindClass( env, "some/class/name" );
```

versus this C++ syntax:

```
JNIEnv *env;

env->FindClass( "some/class/name" );
```

The C++ syntax is quite obviously more readable and less prone to errors, as you'll see even more clearly when we get into the details of the JNI in Chapter 4.

It is actually possible to implement native methods using other languages, like FORTRAN or COBOL, since code written in these languages can be compiled into libraries that contain symbol tables, just like C and C++ programs are. However, the standard NMIs only provide tools to produce interface code in C or C++. To use another language, you have to write an additional glue layer of C or C++ code that calls the FORTRAN or COBOL code. Although this technique is more complex and time consuming, it is often the easiest way to integrate legacy code, especially from mainframe machines.

The Shape of Things To Come

This chapter has outlined some of the salient issues of native methods programming. If I haven't dissuaded you from implementing native methods, you should be praised for your courage! In upcoming chapters, we'll be exploring how to write native methods using both the Java Native Interface and the Raw Native Interface on Unix, Windows 95/98/NT/2000, MacOS, and OS/2 platforms. This material will give you plenty of hints on maximizing native method efficiency for different operating systems and virtual machines.

Pictish Symbol Stones and the Sample Code

Throughout this book, I have endeavoured to use a unified set of examples which will hopefully allow you to figure out what each piece of code does rather than trying to figure out what the example means!

The examples revolve around the archaeological fascination of Pictish Symbol Stones. These monuments, sprinkled liberally around the north and east of Scotland, were created around AD600-AD900 (or thereabouts. No-one's really that sure yet...). They can be partitioned into three "classes", the earliest being rough pillars of stone with strange symbols incised into them, then rectangular slabs with interlace and symbols, and finally slabs with no symbols and only interlace.

Our example system define two Java classes to encapsulate some information about these stones called `SymbolStone` and `Dimension3D`, the latter describing the physical proportions of the stone.

If you wish to find out more about these fascinating stones, or see what they tend to look like, a good place to start is:

```
http://www.symbolstone.org/archaeology/archaeodb/index.html
```

Where using the example system would be really convoluted, I've simply resorted to illustrating various techniques with more pathological examples.

Chapter 3. Java Architecture

This chapter turns towards more technical issues, namely the architecture of the Java Virtual Machine, which we shall refer to from now on as a JVM. A JVM provides a platform-independent and secure execution environment for Java programs. All Java programs are compiled into platform-independent byte codes that are the machine language of the JVM. As a native method programmer, your interaction with Java objects and classes is through the JVM, so it is beneficial to understand the architecture that is implemented by most, if not all, JVMs.

The Java Virtual Machine

The JVM is a representation of an imaginary computer, or, in other words, the JVM is actually a program emulating a computer.[1] This allows us to write programs for this imaginary computer, and, provided that the software that emulates the imaginary computer is running on a particular operating system, we can run those programs on that operating system. Similarly, if the program that emulates the imaginary computer runs on more than one operating system, we can use the same code, without recompilation, on all of those operating systems. This is the concept of the Java Virtual Machine and the root of the "Write once, run anywhere" mantra of Sun.

More importantly, however, is that virtual machines are not restricted to running only as software-based implementations. There are plans in progress to burn a JVM onto a chip that can be embedded into small appliances, such as television sets or even toasters! Even though the hardware is radically different in function to a computer, these consumer devices will be capable of executing exactly the same Java programs, without modification, that run on Windows or Unix machines in software-based JVMs.

The embeddability of the JVM into many different operating systems and applications makes Java quite powerful, but the existence of a JVM alone is not enough to be of any real use. For example, we need a way to input information to and output information from a JVM. What's needed is the ability to load Java code that performs basic functions, in the same way that C programs require a standard library that contains many simple functions that programs use, such as `printf()` and so on. This code is encapsulated as *class files*, which are small Java programs that implement discrete object classes in the object-oriented programming methodology.

Java comes with a number of standard class files, grouped into packages, that provide basic functionality, such as string manipulation, file handling, and networking. While some of these classes are written completely in Java, not all of them can be, due to the fact that the JVM may be running in radically different environments. Consider the code needed to provide I/O facilities when one JVM is running on a standard UNIX workstation with a keyboard and monitor, while another JVM is runnning on a small hand-held video game that has an input device of a joystick and a small LCD for output. Each JVM requires its own underlying I/O handling capabilities, and these are provided by *native* code for that particular platform or operating system.

Native code, as we discussed in Chapter 2, is code written with a compilable language, usually C or C++, that is compiled to a form that is executable only on computers running identical operating systems.[2] The actual guts of the I/O code for Java is generally written in C and compiled into libraries that are made available to the JVM for a particular platform. These libraries are called by the standard Java class files to provide I/O support.

Thus, there are three distinct layers to a functional JVM:

1. The JVM itself, which provides the general execution environment for Java programs
2. A collection of Java programs, or class files, that provide standard Java functionality
3. Native code implementations of platform-specific functions, such as I/O, that translate the desired operations in the Java class files into appropriate functions for the platform upon which the JVM is running

Class Files

Class files are the backbone of a fully functional Java system, since they are the programs that the JVM runs. These files contain the results of compiling Java source code; they are the machine-code representations of particular Java programs. As with machine-code that runs on standard processors, such as the Intel 80x86 family, the processor interprets each byte of a program in a different way. For example, some bytes instruct the processor to store values, and others instruct the processor to start executing machine-code at a different location. As a native methods programmer, you don't need to know about the format of class files, but if you want to know more, you can check out *Java Virtual Machine*, by Jon Meyer and Troy Downing (O'Reilly), or *The Java Virtual Machine Specification*, by Tim Lindholm and Frank Yellin (Addison-Wesley).

Each class file contains the machine-code for a Java *class*, which essentially encapsulates the definition of an *object* within the object-oriented software design methodology. Thus, if we want to create a new object of a given class, the constructor of the appropriate class is executed to create a new instantiation of that class. You can now manipulate the new object in many different ways, the most common being to execute methods on the object.

Many class files are distributed with a JVM; they provide a large base of classes fundamental to the execution of Java programs. For example, the ultimate parent of every class is `java.lang.Object`. This has its own class file, as does the mathematics class, `java.lang.Math`. These classes, along with all the others in the core Java APIs, can be used to create some complex Java programs.

Class files demonstrate a core feature of Java: the flexibility to dynamically extend the language by loading new class files on demand from various places. Through a powerful mechanism called the

`ClassLoader`, classes can be pulled into the JVM from many disparate sources, such as from a local filesystem, over a network, or even across cables connected to the machine running the JVM. This feature makes Java an extremely powerful and customizable tool, but could possibly be exploited to load destructive or unsafe Java programs into a JVM.

To allow the flexibility of dynamic extension but maintain the security of the JVM, the `ClassLoader` performs an extremely important task known as byte-code verification. This subjects the class file to rigorous checks before it is loaded into the JVM. A detailed discussion of this operation is beyond the scope of this book, but there is one aspect of the process that pertains directly to native method programming.

Part of the byte-code verification process checks that the definition of the class, its fields, and its methods are all valid. Further checking is done on the return type of each method to ensure that the byte code returning values does not violate type-safety.[3] The actual test is performed by comparing the type signature associated with a field or method against the type signature expected by the byte-code verifier. A type signature is simply an encoding of a data type (e.g., `int`, `float`). Type signatures also encompass actual Java classes, such as `java.lang.Object` or `java.lang.String`. As a native method programmer, you must, unfortunately, understand the encoding that the JVM uses to represent data types, so that you can fully interact with Java classes and the JVM itself, as we shall discuss in later chapters of the book.

Type Signatures

Type signatures are strings of characters that represent data types, such as `int`, `char[]`, and even `java.lang.String`. The JVM uses these encoded signatures to verify byte codes and, possibly more importantly, to ensure that the correct method is invoked where polymorphic methods exist.

As we'll see later, when you are writing native code and you want to interact with a Java object by calling a method on it, you have to specify exactly which method you want to use by giving the type signature of the method. This is because in Java you can have polymorphic methods. In other words, you can have a single method declared in two different ways. For example:

```
public int someMethod( int a, int b, int c );
public int someMethod( int a, float b );
```

Without type signatures, there is no way to distinguish between these methods, as all your native code can say is "execute `someMethod()`". Which one gets called?

To specify exactly which method you want to execute, you can say "execute the version of `someMethod()` with this particular type signature". This ensures that the correct method is called.[4] With this explanation in mind, let's take a look at the way data types are mapped to type signatures.

Primitive Data Types

Data types in type signatures tend to be represented by single characters, to make them faster to decode and more compact to store.[5] Table 3-1 lists the mapping between primitive data types and their type-signature characters.

Table 3-1. Primitive Type Signature Characters

Character	Type	Description
`B`	`byte`	signed byte
`C`	`char`	character
`D`	`double`	double precision IEEE float
`F`	`float`	single precision IEEE float
`I`	`int`	integer
`J`	`long`	long
`S`	`short`	signed short
`Z`	`boolean`	true or false
`V`	`void`	void

In the type signature for a method, the data types for the method parameters are specified within parentheses, in the order that the parameters are specified. In addition, the return type of the method is listed after the parameter list, outside of the parentheses. Given these rules and the mappings in Table 3-1, our earlier `someMethod()` method declarations can be described in the following way:

```
(III)I
(IF)I
```

Note that the method name is *not* included as part of the type signature for the method. As you can see, type signatures for methods that use primitive data types are quite simple.

Class Types

Because the JVM can load classes dynamically, the encodings for class types cannot be mapped statically within the JVM, as the primitive types are. Fortunately, the developers of the JVM have devised an extremely simple and elegant solution to this problem. To represent a Java class in a type signature, you specify the letter `L`, followed by the fully qualified class name (e.g., full package name plus class name), followed by a semi-colon (`;`).

Consider the following method declaration, in which the method takes two objects as parameters and returns a third object:

```
com.oreilly.returnObject someMethod( java.lang.Object o1,
                                     java.lang.Integer i1 );
```

In this case, the method has the following type signature:

```
(Ljava/lang/Object;Ljava/lang/Integer;)Lcom/oreilly/returnObject;
```

Note that the dots (`.`) used in package names are replaced in type signatures by slashes (`/`). Most operations that directly reference classes in the JVM tend to convert the period character into a slash.

If a class does not belong to a package, the class name stands alone for reference purposes. In other words, a class called `someClass` that does not specify a package (and is therefore in the default package) is mapped to a signature of `LsomeClass;`.

Array Types

The last thing we have to consider with regard to type signatures is array types. When a parameter or a return value is an array, it is specified with a left square-bracket (`[`), followed by the type signature for the data type of the array. For example, a method with a return type of `int[]` has the following signature:

```
()[I
```

For a more complex example, consider a method that takes an array of `char` values as a parameter and returns an array of `Object` values. The signature for this method is:

```
([C)[Ljava/lang/Object;
```

This is all quite straightforward, but what happens in the case of arrays of arrays? All of the dimensions of a multidimensional array must be of the same type, so the rule is to use multiple left square-bracket characters as a prefix, one for each dimension of the array, followed by the type of the array. Thus, a method that takes a `char[][]` as a parameter can be expressed as:

```
([[C)
```

Native Code Libraries

As I explained earlier, the JVM relies on natively implemented code to provide its essential I/O functionality. But how exactly is this native code packaged up and made available to the JVM? And more importantly, how do you package up native methods that you have written and make those available to the JVM?

The answer lies in the ability of modern operating systems to dynamically load libraries of information into running programs on demand. This is a similar idea to that of loading class files as we need them. In C and C++ programming, it is quite common to group many small routines that perform similar tasks together into an entity called a *library*. For instance, the math library that exists on all Unix platforms has functions that calculate the sine, cosine, and tangent of angles, among other things. Each of these functions is really too small to have in its own separate file, but together they form a useful collection. When a developer wants to use the math functions, she simply *links* her application with the math library. Native code implemented for a JVM gets similar treatment.

The instructions for compiling a library for a particular operating system obviously depend on the operating system. Compilation also depends on the native method interface you have chosen for implemeting your native methods, as you'll see when we get to covering the two native method interfaces in the remainder of this book. Once you have compiled your library for use in the JVM, however, you have to place it somewhere on the filesystem, so that the JVM can find it when it wants to load it. Again, this is a platform-specific and JVM-specific issue. The following sections offer some guidance on compilation and installation for the most popular setups.

Unix/Linux

Unix-based platforms, such as Solaris, Irix, and Linux, all use a piece of software called a dynamic linker or dynamic loader. The actual program is usually called *ld.so*. This software is invoked when a program containing libraries is loaded; its purpose is to load all the libraries into the running executable to "complete" the program. Therefore, to allow *ld.so* to handle loading the library that contains your native method implementation, you simply need to configure that program correctly.

ld.so is generally configured with environment variables, such as `LD_LIBRARY_PATH` or `LD_RUN_PATH`. These variables contain lists of directories that are searched for dynamic libraries when a program instructs *ld.so* that a certain library is required. For example:

```
/usr/lib:/lib:/usr/local/lib:/usr/local/java/lib/i586
```

There may be slight differences in the way in which *ld.so* operates on a particular Unix platform, so it is always wise to read the *man* page for it and take the appropriate steps described therein.

By default, libraries under UNIX are named *lib*`somename`*.so*. In other words, a library is prefixed with *lib* to indicate that it is a library, and suffixed with *.so* to specify that the file in question is a shared library. When you are referencing the library, however, either at compile-time or when you are loading it into the JVM, you don't need to mention either the *lib* prefix or the *.so* file extension.[6]

When you are building a shared library under Unix, you have to tell the compiler that you are building a shared library, not a complete program. If you have not specified the appropriate compiler flags, you may see a message about the `main()` function being missing. For example:

```
/usr/lib/crt1.o(.text+0x5a): undefined reference to 'main'
```

The flags that specify the creation of a shared library vary on a per-platform and per-compiler basis. Let's assume we want to create a library called *libHelloWorld.so* from a file containing native methods called *nativeHelloWorld.cpp*. Table 3-2 shows some of the more common incantations required to do it.

Table 3-2. Unix Shared Library Compiler Incantations

Platform	Command
Linux	`gcc -shared -o libHelloWorld.so nativeHelloWorld.cpp`
Solaris	`CC -G -o libHelloWorld.so nativeHelloWorld.cpp`
Irix	`CC -shared -o libHelloWorld.so nativeHelloWorld.cpp`
AIX	`xlC_r -c nativeHelloWorld.cpp`
	`makeC++SharedLib -p 1 libHelloWorld.so nativeHelloWorld.o`

This list is by no means exhaustive; you should *always* check the *man* page for your compiler to see exactly which flags are needed.

When you are compiling native libraries for the Sun JVM, you also need to make sure you are referencing the appropriate include-file directories for the JVM, so the compiler can find files like *jni.h*. In particular, you need to reference both the generic Java SDK include directory and the platform-specific one, or the compiler may give you errors about being unable to locate *jni_md.h*, among other things.

The include file directory is always called *include* and exists within the base Java installation directory on your machine. This directory could be literally anywhere on your filesystem, depending on whether you installed Java yourself or it came preinstalled from your machine vendor. The easiest, but not necessarily best, way to locate the *include* directory is to use the *find* program and search for a file named *jni.h*.

The platform-specific directories vary by operating system, but they are always located within the generic *include* directory. Table 3-3 shows some of the more common directories as you would specify them with compiler flags.

Table 3-3. Unix Include File Locations

Platform	Compiler Flags
Linux	`-I/usr/local/java/include` `-I/usr/local/java/include/genunix`
Solaris	`-I/usr/local/java/include` `-I/usr/local/java/include/solaris`
Irix	`-I/usr/java/include` `-I/usr/java/include/irix`

Typing all this stuff in is fairly tedious, so Unix gurus tend to use the *make* program to compile libraries. Here's a sample *Makefile* for building a "Hello World" example for Linux, Solaris, and Irix:

```
# The source code containing the native method implementations
SRCS=nativeHelloWorld.c
OBJS=$(SRCS:.c=.o)

# The Java source code of the application
JSRCS=HelloWorld.java
JOBJS=$(JSRCS:.java=.class)

# The location of any include files used by the native methods
# You would need to alter “genunix” to suit your own platform,
# e.g.,
#       Solaris -> ’solaris’
#       Linux   -> ’genunix’
#       Irix    -> ’irix’
INCLUDES=-I$(JAVA_HOME)/include -I$(JAVA_HOME)/include/genunix -I.

# The flag used to generate shared libraries. This needs to be
# altered to suit your platform, e.g.,
#       Solaris -> ’-G’
#       Linux   -> ’-shared’
#       Irix    -> ’-shared’
SHARED_FLAG=-shared

.SUFFIXES: .java
.SUFFIXES: .class

# How to compile the Java code
```

```
JAVAC=javac -verbose -classpath $(CLASSPATH)

# How do we generate the include files for the Java classes?
JAVAH=javah -jni

all: $(JOBJS) $(OBJS) libHelloWorld

# Compiles the Java code if necessary and generates the include file
.java.class:
$(JAVAC) $*.java
    $(JAVAH) $*

# Compiles the C code into an object file
.c.o:
gcc -c $*.c $(INCLUDES)

# Creates the shared library
libHelloWorld:
gcc $(SHARED_FLAG) -o libHelloWorld.so $(OBJS)

# Removes all the intermediate build files
clean:
rm -f *.o
rm -f *.class
rm -f *.so
```

This *Makefile* is pretty simplistic, but it does compile your Java code for you when the source file is newer than the object file or shared library. It also compiles your native method implementation file when you have changed that. Finally, it creates the shared library containing the native method implementations required for your application to run.

Windows

Windows systems (95, 98, 2000, and NT) are similar to Unix systems in terms of how they load libraries, known as DLLs under Windows. Windows searches directories for suitably named libraries to provide missing symbols for programs, but unlike with Unix, there is no special piece of software that performs this task. To point Windows at a directory that contains shared libraries that need to be loaded by the JVM, you simply add that directory to the `PATH` environment variable.

In terms of filenames, dynamic libraries on Windows systems have *no* prefix, but they are suffixed with *.dll*. Thus, a shared library named *libHelloWorld.so* on a Unix system appears as *HelloWorld.dll* in

Windows. But from Java's perspective, this file is still simply *HelloWorld*. This translation between names is performed automatically by the JVM.

This situtation is generally fine for loading libraries with the JVM released with the Microsoft Java SDK or the Sun Java SDK or JRE packages. However, Netscape Communicator also requires that the library be installed within its installation area, typically *\Program Files\Netscape\Communicator\Program\Java\Bin*, since the `PATH` environment variable is regarded as untrusted.

Building native libraries with an IDE, such as Visual C++, is extremely straightforward. When creating a new project, you should select the project type of *Win32 Dynamic-Link Library*. This sets up the appropriate default settings in your workspace for compiling a DLL.

For a JNI native library, the only other setting you need to configure involves the include file directories. Bring up the *Project Settings* dialog and input the directories where the Java include files are stored on your system. By default, these files are installed into *\jdk1.2\include* for the Java SDK 1.2 release. You must also add the platform-specific include files directory, to avoid getting strange errors, such as being unable to locate *jni_md.h*. For Windows platforms, this directory is *\jdk1.2\include\win32*.

Setting up compilation options for native methods written using RNI is somewhat more involved. You have to configure the include-file path as with JNI, to locate the directory that contains the main RNI implementation include file, *native.h*. If you have installed Version 4.0 of the Microsoft Java SDK, this location is *\Program Files\Microsoft SDK For Java 4.0\include* by default. In addition, for RNI builds, you must link with a library provided by Microsoft that contains the implementations of the RNI functions. This library is called *msjava.lib* and can be found in the *\Program Files\Microsoft SDK For Java 4.0\lib\i386* directory for Intel-based platforms. This file should be added to the *Link* tab in your project settings.

After configuring these options, you should be able to compile your native method implementations into a shared library suitable for use with either JNI- or RNI-based JVMs. I personally create a different configuration within my workspace for each JVM, which allows me to build suitable native libraries quickly and easily. For example, I might have configurations for *Sun VM*, *Sun VM Debug*, *Microsoft VM*, *Microsoft VM Debug*, *SuperCede VM*, and *SuperCede VM Debug*, so that I can not only build separate libraries, but batch build all of them for distributions of code. The advantage of this approach is that once each project is set up, I don't need to make any alterations to recompile for a different JVM.

OS/2

OS/2 tips its hat to both Windows and Unix in the way in which it searches for native libraries. OS/2 uses a configuration file called *config.sys* boot-time to set up various environment variables that regulate how the operating system runs. This is similar to the *autoexec.bat* file under DOS and Windows. Within this file, the `LIBPATH` variable specifies a list of directories that contain libraries that can be dynamically loaded by programs. Thus, you should add the directories that contain your shared libraries to this environment variable and reboot.

Libraries on OS/2 are named just like libraries on Windows, in that the library name is suffixed with *.dll* and there is no prefix. Thus, if you want to load a library called *HelloWorld* from Java, the filename under OS/2 is *HelloWorld.dll*.

That's the easy part. Compiling native method implementations into a shared library can be a bit trickier. I use the EMX/GCC package on OS/2 to compile programs, instead of a commercial compiler such as IBM's Visual Age, since EMX tends to make porting from Unix a lot easier in terms of recognizable features. Building native methods under Visual Age is probably more straightforward though, since it's a GUI-based IDE, rather than a collection of a compiler, a linker, and other tools.

Compiling the native method files is easy, in that you simply use *gcc* in the standard way, by specifying the C or C++ file to compile and any include-file paths. For example, the following command entered at an OS/2 prompt compiles the *nativeHelloWorld.cpp* file:

```
gcc -c nativeHelloWorld.cpp -I/java11/include -I/java11/include/os2 \
    -DJAVA_EXE -DOS2 -Zomf
```

There are a couple of strange flags here that need some elucidation. The JNI include files shipped by IBM with their port of the Sun JVM have some conditional compilation sections that are picked up when the `OS2` value is defined. To ensure that these OS/2-specific sections are correctly picked up, it is recommended that you define the `OS2` symbol. Also, the JNI function-calling convention can differ depending on which compiler you are using. In order for EMX to work correctly, you need to define the `JAVA_EXE` symbol to ensure that the correct values of `JNIEXPORT` and `JNICALL` are used.[7]

The final flag worthy of special discussion is `-Zomf`. With EMX, there are two different methods of linking object files together. One method is to use a port of the GNU linker called *ld* that produces libraries. *ld* accepts the standard object file output format as input files to link. However, the use of *ld* can be problematic when shipping shared libraries to other machines. The other linker, shipped with OS/2, is called LINK386. This linker accepts OMF format object files and can be used to generate native libraries that are easily shipped to other OS/2 machines. Specifying the `-Zomf` flag tells *gcc* to produce OMF object files as opposed to files suitable for linkage with GNU *ld*. These files are suffixed with a *.OBJ* extension, as opposed to the more standard *.O*.

Once you have compiled all of your files that contain native method implementations, you need to link them together. Once again, *gcc* can be used in the following manner:

```
gcc -Zso -Zdll -Zomf -Zsys nativeHelloWorld.obj HelloWorld.def \
    -o HelloWorld.dll
```

The flags prefixed with `-Z` tell *gcc* to use LINK386 to produce a shared library. They also indicate that the shared library must not depend on EMX at runtime. In other words, if you give the DLL to another programmer, that programmer does not need to have EMX installed to use the DLL.

The other interesting aspect of the linkage step is the file named *HelloWorld.def*. This file is not a compiled file at all. Instead, it specifies a list of functions contained within the source files that should be exported, or made available, outside the shared library. Every native method that you have implemented in any of the source files must be listed in this file, along with some other directives that are used by LINK386 to determine exactly how the library should behave. Here is a sample export file that can be used as a template, simply changing the exported function names:

```
LIBRARY INITINSTANCE
DATA MULTIPLE NONSHARED
CODE SHARED
EXPORTS
    Java_HelloWorld_printString
    Java_HelloWorld_anotherNativeMethod
    Java_SymbolStone_describe
    Java_Dimension3D_getWidth
```

The final issue regarding compilation of shared libraries under OS/2 is automating the build. You can download a port of GNU *make* from *http://www.leo.org*. This allows you to use a *Makefile*, as described earlier for Unix systems.

MacOS

Native method programming on MacOS is fundamentally similar to native method programming on other platforms. The main problems you tend to encounter revolve around trying to configure your build environment correctly and diagnose why a particular native library is failing to load. Such is MacOS!

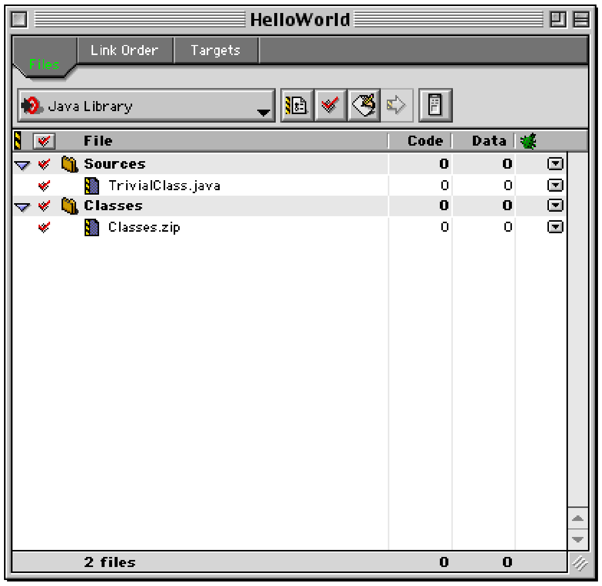
To make things a little bit more complicated, the Mac supports a few different JVMs, such as Apple's MRJ (Mac Runtime for Java) and Metrowerks' JVM, bundled with their CodeWarrior product. Since both of these JVMs are actually quite commonly used and as you'll probably be using CodeWarrior to compile your code anyway, I'm going to discuss how to use native methods with both of them.

Libraries on MacOS are known as shared libraries and have no prefix. They are usually located within the *System Extensions* folder and have a creator type of *shlb* and a creator code of *cfrg*. The naming of shared libraries differs depending on which JVM you are using. For example, if you want to load the *HelloWorld* library from Java, the Apple MRJ expects a shared library name of *HelloWorld*. CodeWarrior is actually bundled with two JVMs, a standard one and a JIT-aware one. These JVMs expect different library names: *Java_HelloWorld* and *JavaJIT_HelloWorld*, respectively.

Armed with this knowledge, the next step is to actually compile your native method implementations into shared libraries that can be loaded into the JVMs. The following discussion uses CodeWarrior Professional Release 4, although other IDEs under MacOS should be similar enough for you to work out what's going on. The first step is to create a new project for our sample application. We'll call this HelloWorld. Within CodeWarrior, you should choose to create a new *Java Library*. When you click *OK* in the *New Project* window, a new project window is opened for the HelloWorld project.

By default, the new project window isn't very useful for working with native methods. I'd recommend making some changes to the project groups, so that the window looks like the one shown in Figure 3-1. This format allows you to quickly see which external native libraries and classes are being used and also which Java and C++ source code files your project composed of.

Figure 3-1. A Configured HelloWorld Project

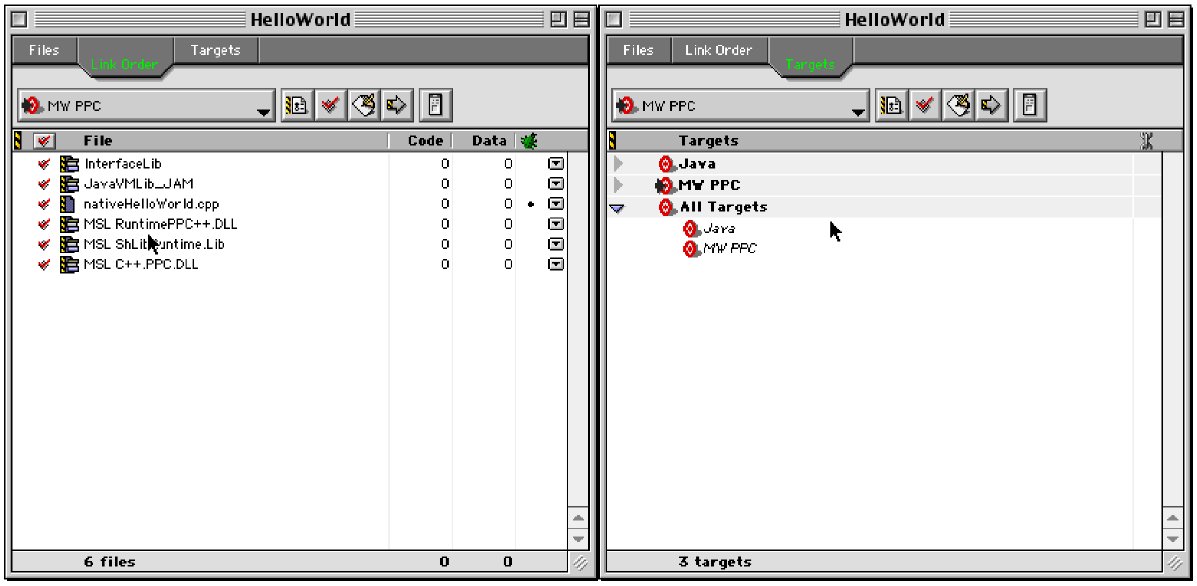


File	Code	Data
Sources	0	0
TrivialClass.java	0	0
Classes	0	0
Classes.zip	0	0
2 files	0	0

This project currently has three targets. The *Java*target is defined to build the Java source files. The *MW PPC* target builds the shared library suitable for use on a PPC-based Mac under the Metrowerks JVM. *All Targets* is simply there to allow us to build everything in one go. We will be adding extra targets for the Metrowerks JIT JVM and the Apple MRJ shortly.

The files that you see in the *Native Libraries* group can be found in the CodeWarrior installation on your machine. These are required for basic MacOS system functions and the JNI to work; they must be specified in the order shown. Furthermore, you should correctly set up the link order for your native code targets as shown in Figure 3-2.

Figure 3-2. The Appropriate Link Order for HelloWorld

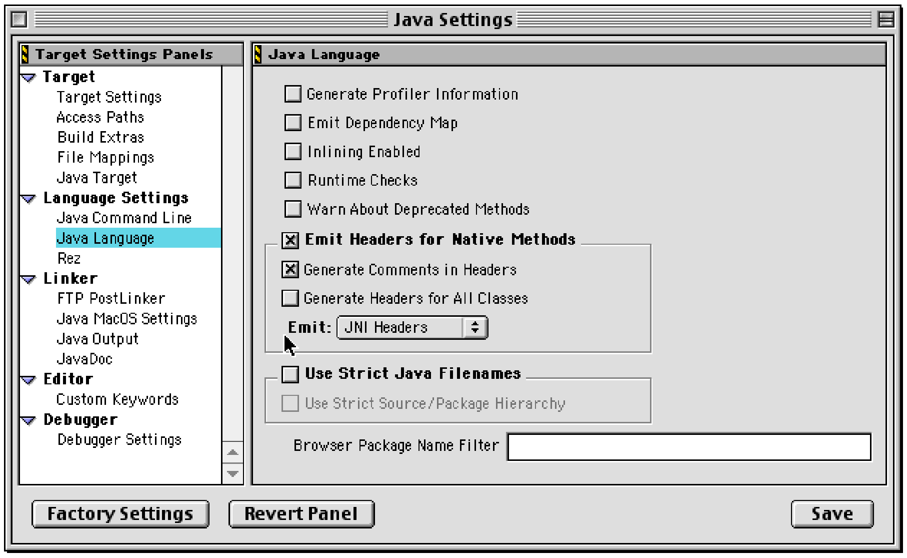


File	Code	Data
InterfaceLib	0	0
JavaVMLib_JAM	0	0
nativeHelloWorld.cpp	0	0
MSL RuntimePPC++.DLL	0	0
MSL ShLibRuntime.Lib	0	0
MSL C++.PPC.DLL	0	0
6 files	0	0

```
Targets
Java
MW PPC
All Targets
    Java
    MW PPC
3 targets
```

Now we need to configure each target to build the various bits and pieces of code. The first step is to configure the Java target to have a linker type of *Java Linker*. This ensures that your Java code is built correctly. You also need to configure CodeWarrior to automatically generate include files for your JNI methods, as shown in Figure 3-3. This replaces the *javah* tool used on other platforms. The other settings required for Java compilation can be found in the CodeWarrior documentation; they don’t relate directly to configuring CodeWarrior for native code compilation.

Figure 3-3. Configuring CodeWarrior to Emit Headers



Java Language

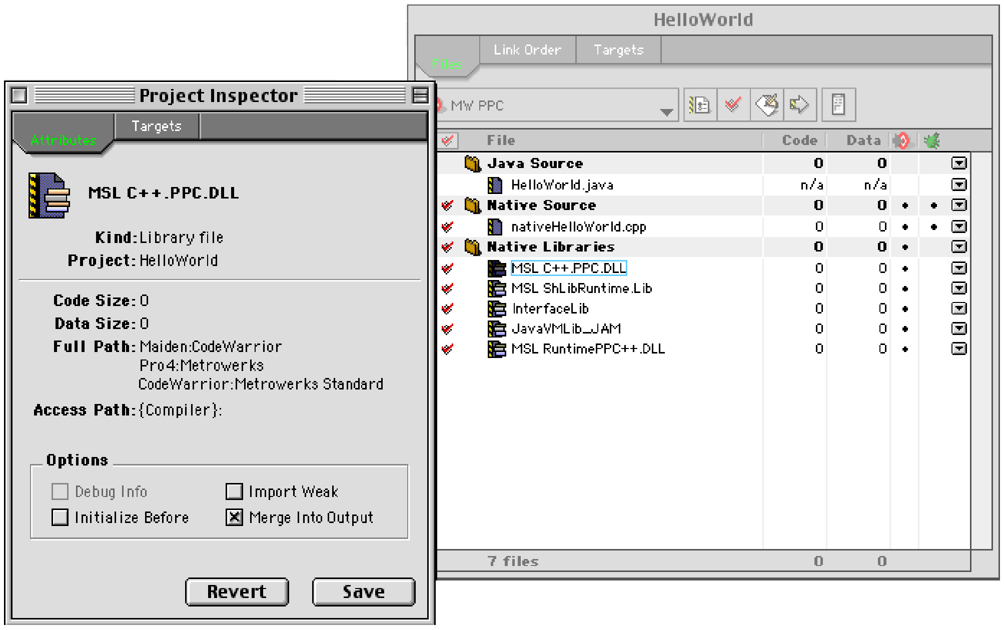
- [ ] Generate Profiler Information
- [ ] Emit Dependency Map
- [ ] Inlining Enabled
- [ ] Runtime Checks
- [ ] Warn About Deprecated Methods
- [✓] Emit Headers for Native Methods
  - [✓] Generate Comments in Headers
  - [ ] Generate Headers for All Classes
  - Emit: JNI Headers
- [ ] Use Strict Java Filenames
  - [ ] Use Strict Source/Package Hierarchy
- Browser Package Name Filter:

The next step is to configure the *MW PPC* target to build a shared library suitable for loading into the JVM. This target should have the linker type of *MacOS PPC Linker*.[8] In the *PPC Target* setting, be sure to specify the name of the shared library. Because this target is building a shared library for use with the Metrowerks JVM, the shared library name is simply *HelloWorld*. Similarly, in the *PPC Linker* setting, you must set the *Segment Name* to be the name of the shared library. The final target configuration task is to clear any unnecessary linker settings, such as entry points into the shared library. No entry points are required and specifying any will cause your library linking to fail.

In addition to compiling your native code, you also need to generate an export file that lists all the functions that your shared library exports to applications that are using the library. The export file must contain the names of all your native methods. CodeWarrior can automatically generate this file for you if you tell it to. Once you have generated this file, I recommend that you include the file explicitly as a target file to compile and switch off auto-generation. You may also need to edit the file to remove any spurious symbol definitions, such as static variables used in your native code. The file is a simple list of function names in ASCII form, so this is easy to do.

The last configuration task you may need to perform is to have the MacOS libraries you are using within your project imported into the shared library. If you don't do this, it is unlikely that these libraries will be found when you are loading your shared library from Java and an `UnsatisfiedLinkError` may be thrown. Configuring the libraries is easy. Simply click on a library name and select *Project Inspector*. A window similar to that shown in Figure 3-4 appears and you can click on the *Merge Into Output* option.

Figure 3-4. Merging Shared Libraries



MSL C++.PPC.DLL

Kind: Library file
Project: HelloWorld

Code Size: 0
Data Size: 0
Full Path: Maiden:CodeWarrior Pro4:Metrowerks CodeWarrior:Metrowerks Standard
Access Path: {Compiler}:

Options

- [ ] Debug Info
- [ ] Import Weak
- [ ] Initialize Before
- [x] Merge Into Output

	File	Code	Data		
	Java Source	0	0		
	HelloWorld.java	n/a	n/a		
✓	**Native Source**	0	0	•	•
✓	nativeHelloWorld.cpp	0	0	•	•
✓	**Native Libraries**	0	0	•	
✓	MSL C++.PPC.DLL	0	0	•	
✓	MSL ShLibRuntime.Lib	0	0	•	
✓	InterfaceLib	0	0	•	
✓	JavaVMLib_JAM	0	0	•	
✓	MSL RuntimePPC++.DLL	0	0	•	
	7 files	0	0		

Once you have done these things, you should be able to compile your Java and native code, producing a JAR file and a shared library. At this stage, the shared library can be moved into the *System Extensions* folder where it is loaded by the JVM. Adding support for additional JVMs or native code targets is just as straightforward. These can be created initially as copies of the already configured target and tweaked accordingly. The most important setting is the name of the shared library; it should be set according to the rules I outlined earlier in this section.

Loading Native Code into the JVM

Now that you have a library containing native code on your machine, you've put it in the correct location, and you've configured the appropriate operating system parameters to know where to find the library, you need to instruct the JVM to load the native code. This is actually the most straightforward procedure in the whole project: you simply need to invoke the `loadLibrary()` method defined in the `java.lang.System` class. This works for all JVMs except Netscape Navigator, which implements a more sophisticated security management system, as we'll discuss later in this section.

The `System` class encapsulates platform-specific information about both the machine and the runtime that you are using (e.g., the `System` class for the JVM shipped with Sun's Java SDK is different from the `System` class shipped with Microsoft's JVM, even though they may be running on the same computer). Therefore, to load the library in a platform-independent way, we need to load it from Java itself. The following code snippet illustrates the procedure, assuming our native library is called *HelloWorld*:

```
/* Loads a native library called HelloWorld */
public class testHelloWorld {

    static {
        System.loadLibrary( "HelloWorld" );
      }
  }
```

We have placed the `System.loadLibrary()` invocation in a `static` block that is executed when the `testHelloWorld` class is loaded in the JVM. You don't need to add the platform-specific prefixes or suffixes to the library name (e.g., *lib*, *.so*, *.dll*). The JVM automatically works out the appropriate naming scheme for the libraries on the machine upon which it is running.

What happens if something goes wrong? Well, most problems occur due to unresolved symbols in your native code (i.e., functions or variables that you have referenced, but never defined). This is not an unusual occurrence, as your code can compile just fine, but you won't discover that a function definition has mismatched types or that you've made a typo until you try to link the library to another program. A general purpose `Exception` called `UnsatisfiedLinkError` is thrown whenever a shared library either does not exist or has unresolved symbols within it.

Fixing problems inherent with unresolved symbols can be quite a daunting process and is generally specific to your particular development environment. Here's a tip I find useful on Unix systems: run the *nm* utility against your library and *grep* for the letter "U" or word "UNDEF", to see which symbols are undefined. In many cases, undefined symbols are legitimately undefined, since they actually exist within another library, like the math library. But you may be able to spot some obvious problems with this technique. Another multipurpose tip is to use the highest level of compilation and link debugging available to you within your development environment, as this may flag suspicious problems that can cause your library to contain unresolved symbols.

If your library has simply not been found, you should double check that you have correctly set the appropriate environment variables and that the library has the correct name. Another Unix tip is to use the *truss* or *strace* programs to trace all system calls while the JVM is running.[9] You can *grep* the copious output of these programs for the name of your library to see how the particular call to perform the load operation failed. This might help you narrow down the cause of the failure somewhat.

To make your Java code more user-friendly in cases where the library containing your native code has failed to load, you should catch the resulting `UnsatisfiedLinkError` and present a pleasant error message to the user, instead of spewing gobbledigook onto the screen. The following code illustrates a simple mechanism for doing this:

```
/*
 * Loads a library named HelloWorld with some sort of error
 * checking.
 */
public class testHelloWorld {

  static {
    try {
      System.loadLibrary( "HelloWorld" );
    } catch ( UnsatisfiedLinkError e ) {
      System.err.println( "The program has failed to load an integral" );
      System.err.println( "component of the system. Please contact the" );
      System.err.println( "Java development team and give them the" );
      System.err.println( "following information:" );
      System.err.println( "\n\t-----------------\n" );
      e.printStackTrace();
    }
  }
}
```

And that’s all there is to loading native code into the JVM!

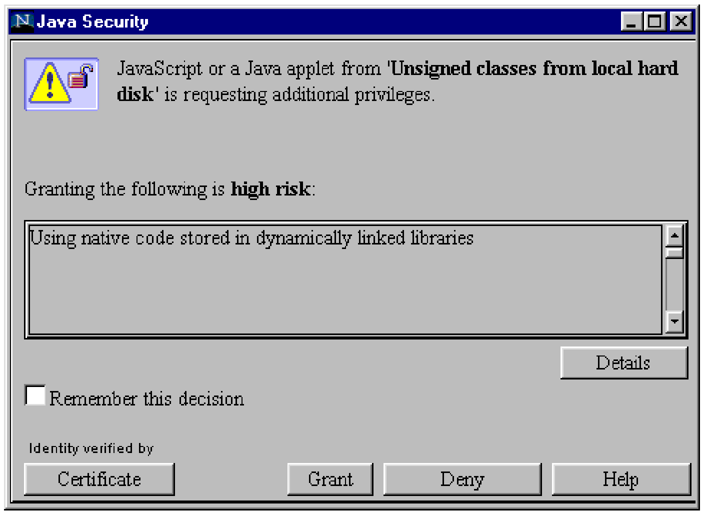
If you are using native code within an applet, you should know about an important restriction concerning applets: applets themselves cannot call `System.loadLibrary()` since this is regarded as being unsafe. However, applets can instantiate other classes that load native libraries, but these “helper” classes must reside in the user’s `CLASSPATH` (i.e., they must be installed on the user’s machine). Also note that native libraries bundled into JAR files are not be loaded by applets due to security restrictions.

Netscape and Native Libraries

Netscape Navigator implements a slightly tighter security layer regarding native libraries than the Sun JVM, *appletviewer*, and the Microsoft JVM. With Navigator, you cannot load native libraries unless either the applet has been signed with a privilege that allows it to do this or you specifically request this privilege and the user grants it. For example, if your applet tried to execute the earlier code to load the *HelloWorld* library under Navigator, it would simply fail. Note that Internet Explorer does not implement anything beyond the standard security policy of not allowing applets to call `loadLibrary()`.

We need to use Navigator’s security mechanism to tell the browser that it’s okay to load a library. Performing this is quite easy, using a class called `netscape.security.PrivilegeManager` that is bundled with Netscape Navigator. This class allows you to enable various privileged operations, such as writing to a file from an applet or loading native libraries, but when the privileges are enabled, the browser pops up a window, similar to that in Figure 3-5, asking the user if it's okay to proceed with that untrusted operation.

Figure 3-5. Netscape Navigator's Security Interface



Java Security

JavaScript or a Java applet from '**Unsigned classes from local hard disk**' is requesting additional privileges.

Granting the following is **high risk**:

Using native code stored in dynamically linked libraries

The privilege we need to enable is called `UniversalLinkAccess`. It can be enabled by the following segment of code, which should be executed prior to the `System.loadLibrary()` call:

```
netscape.security.PrivilegeManager pm =
    new netscape.security.PrivilegeManager();
pm.enablePrivilege( "UniversalLinkAccess" );
```

Once the user has allowed the privilege to be enabled, Navigator loads the shared library containing the native method implementations.

Notes

1. Just as, say, a Sinclair ZX Spectrum emulator is a piece of software emulating an actual computer. Given that a Spectrum emulator exists for UNIX and Windows 95, you can now play the same

original Spectrum games on two different operating systems.

2. This is roughly true. Operating systems may run on different processor architectures that are incompatible, whereas compiled programs for different operating systems, but for an identical processor architecture, may run under another operating system. Linux is a good example of this, in that SCO Unix binaries for the Intel 386 processor may run under Linux on Intel 386 processor machines.
3. Since the Java compiler would have flagged this error of type-safety at compile-time, the byte-code verifier can safely assume that the class file has been either corrupted or tampered with.
4. C++ also uses type signatures to distinguish between methods. Unfortunately, to allow C++ and standard C programs to link together, C++ methods can be declared as `extern "C"`, which effectively removes the type signature. This can cause symbol clashes in overloaded method definitions.
5. Not to mention harder to understand!
6. HP/UX differs by using the suffix of *.sl* instead of *.so*. To quickly determine which suffix is used on your system, you can look in */usr/lib* and see whether you see *.so* or *.sl* files. Libraries suffixed with *.a* are not shared libraries so pay no attention to those.
7. These two values are discussed in more detail in Chapter 4.
8. I am assuming that you are using a PowerPC-based Mac. The Apple MRJ does not support the 68K architecture at all.
9. Available on Solaris and Linux, respectively. Remember to redirect both *stdout* and *stderr* to a file, as these programs generate a *lot* of output.

Chapter 4. Introducing the Java Native Interface

The Java Native Interface, or JNI, represents a codification of native-method programming interfaces implemented in various virtual machines. The JNI allows you to write Java native methods using a unified programming API, instead of writing code for the native-method interface implemented in each JVM. Historically, there are three main branches of JVM, each sporting its own native method interface. These include JVMs derived from the original Sun JVM, Netscape's JVM and Plugin Development Kit, and Microsoft's JVM. Without the JNI, you'd be looking at writing at least three separate versions of the same code for your native methods to work on the major JVMs.

The JNI provides a generic programming interface to the JVM, ensuring that code written for the JNI works on any JVM that supports the JNI. Of course, there are penalties to pay for this portability. The most obvious potential penalty is that operation of the JNI may be somewhat slow, since it provides a generic, non-optimized access path into the JVM. Fortunately, most JVMs that support the JNI use optimized access paths and perform internal caching to help route function calls efficiently, so performance is not necessarily a problem, as we'll discuss in the next section.

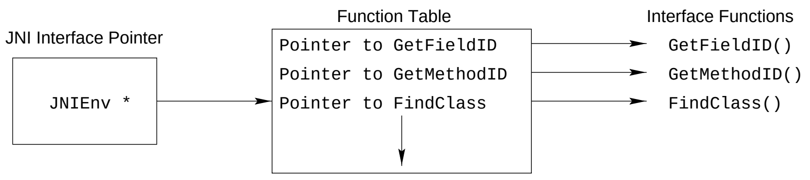
The JNI provides a layer of abstraction over the internals of the virtual machine. This can be construed as a second penalty, especially if you are used to the power of the JDK-1.0 or RNI interface. This is a potentially sound objection to using the JNI, since you now have no direct access to the C `struct` that represents a class. The JNI requires much higher processing overhead to reference the same amount of data; we'll discuss this issue in more detail later in the chapter.

Thus, using the JNI *may* be slightly slower than using an earlier, non-standard native interface. Given that native methods are supposed to be used in handling non-trivial amounts of computation, however, the performance overhead in entering the native method may be negligible compared to the amount of computation the native code performs. In the words of Michael Abrash: "Profile before you optimize." Of course, the benefit of being able to write and maintain a single body of code that works seamlessly on all JVMs is a seductive lure.

The JNI Execution Environment

The main advantage of the JNI is that it is portable across all JVMs, regardless of who has written the JVM. The JNI achieves this by implementing a three-tier access path from your native code to the JVM currently being used, as illustrated in Figure 4-1.

Figure 4-1. The JNI Execution Environment



This figure shows how the JNI has been designed to support not only native methods, but also the ability to embed a JVM within another application. Some good examples of this are the JVMs embedded in the Netscape Navigator and Microsoft Internet Explorer web browsers. The ability to embed a JVM implies that a "host" program, such as a browser, can have multiple Java programs running simultaneously, each of which is a completely separate entity.

To enforce Java security restrictions, programs must be kept completely separate from each other. The JNI enforces this separation with an explicit pointer to the thread within the JVM instance that is currently running the program in question. In JNI terminology, this pointer to the JVM is called the *JNI interface pointer*. Pointers of this type vary on a per-thread basis. In other words, if you invoke a native method from within a single program, but from two different threads, the interface pointer will vary. This keeps thread data compartmentalized, ensuring a higher level of data integrity and security, not to mention optimized performance on multiprocessor systems.

The interface pointer itself points at an array of functions, known as a *function table* or *jump table*.[1] This table simply contains pointers to JNI functions, which are known as *interface functions* and are implemented within the JVM. This design is portable, as any JVM implementation need only implement the standard JNI interface pointer and function table code to allow any native method code to be portable between JVMs. The underlying interface functions within the JVM can then be implemented in a way that provides optimal access to the desired JVM functionality.

Another advantage of this approach is that a virtual machine implementation can have multiple JNI implementations. For example, you might want to have one implementation that does very strict compile-time checking and valid parameter checking, for debugging purposes, and a second that has no checks whatsoever, for release-quality software. With the architecture of the JNI, changing implementations is as easy as swapping the JNI interface pointer to point at the appropriate function table.

Of course, elegant as this design may be, it does add an additional two layers of computation to the invocation of any JNI functions. Over a period of many invocations, this may amount to quite a large amount of time spent just locating the interface functions. But before you worry too much about the performance implications, you should be sure to profile the amount of time spent invoking the JNI functions against the time it takes to execute your native code. The ratio may be inconsequential, in which case you don't need to fret.

When you are writing native code, you don't need to know anything about the function table itself, since all JNI function invocations are funneled through the interface pointer automatically. The interface pointer is of type `JNIEnv *`. It points at a structure that contains pointers for all the JNI functions, where each pointer references the corresponding entry in the function table. The interface pointer is of foremost importance when you are writing native methods using the JNI; it is passed as the first argument to *every* JNI function.

Furthermore, when you want to invoke a JNI function, you must do so through the interface pointer, since the function can only exist within the context of the interface pointer's jump table. The actual use of a JNI interface pointer to locate JNI functions is somewhat more convoluted than you might expect, however.

Since the JNI interface pointer contains another pointer, you must do *two* dereferences to locate the correct jump table. For example, to access the `GetFieldID()` JNI function with a given JNI interface pointer called `env`, you need to write the following C code:

```
JNIEnv *env;

jfieldID someFieldID =
    (*env)->GetFieldID( env, clazz, name, signature );
```

This code is not terribly intuitive, and it can be a source of misery, if you forget to dereference `*env` and dereference `env` instead.

With C++, the situation is a bit better, as you do not need to worry about the extra dereference. The JNI interface pointer is implemented as a C++ class, and the jump table of JNI functions is implemented as a set of C++ member functions within that class, which makes your life a lot easier. The following C++ code stub is functionally equivalent to the previous C example:

```
JNIEnv *env;

jfieldID someFieldID =
    env->GetFieldID( clazz, name, signature );
```

The message here is that if you are implementing your JNI code using C, you need to be *very* careful. Forgeting the extra dereference gives you perfectly legal C code that your compiler will quite happily turn into native code, albeit native code that will inexplicably crash!

Since the C++ JNI interface is more intuitive and easier to debug, the example code in this chapter uses C++. This should not pose any problems for compilation on Microsoft Windows or modern Unix platforms, as C++ compilers, such as *g++* from the Free Software Foundation, are prevalent and easily acquired.

One final issue about language choice arises if you are porting software to a Win32 platform using Visual C++. In this case, you are required to implement your JNI code using the C++ interface, instead of the C interface, due to the automatic definition of various C preprocessor symbols, such as `cplusplus`. The definition of this symbol causes the JNI include files to use the C++ interface automatically, meaning that perfectly legitimate C code will completely fail to compile.

Implementing JNI Native Methods

There are three basic tasks involved with writing native methods using JNI:

- Declaring the appropriate methods as `native` methods in the Java class.
- Writing the native code itself, using the correct function prototypes.
- Providing the “glue” layer that stitches the native code and the Java class together, to ensure that both pieces know where each other are and can communicate.

Writing The Java Code

Informing the JVM that a method is implemented natively, instead of in Java, is the easiest part of developing native methods. You simply declare your native methods with the `native` keyword. Example 4-1 shows a Java class that defines a native method that prints output on the screen. The example prints two messages, one using Java code to print to `System.out`, and the other using the native method.

Example 4-1. Native method declaration (HelloWorld.java)

```
public class HelloWorld {

    /** static initializer block which loads the native library */
    static {
        try {
            System.loadLibrary( "HelloWorld" );
          } catch ( UnsatisfiedLinkError e ) {
            System.err.println( "Cannot load HelloWorld library: " +
                                e.toString() );
          }
      }

    /** The natively implemented method to print a String */
    public native void printNative( String str );
    /** The Java implemented method to print a String */
    public void printString( String str ) {
        System.err.println( "String printed: " + str );
      }

    /** Create a main() method so that we can run this class from a shell */
    public static void main( String argv[] ) {
        /** Create a new instance of the HelloWorld class */
        HelloWorld hw = new HelloWorld();
      }

    /** Construct a new instance of HelloWorld */
    public HelloWorld() {
        /** Print from Java */
        printString( "This string will get printed from Java!" );

        /** Print from native code */
        printNative( "This string will get printed from native code!" );
      }
  }
```

Besides the method that is declared as `native`, there is one other important aspect of native methods programming in this short example. You must remember to explicitly load the library that contains your native methods. If you forget to do this, the invocation of any native method will fail, throwing an `UnsatisfiedLinkError` in the process.

As you can see from the previous example, native methods are invoked in exactly the same way as normal, Java-implemented methods. The advantage of this approach is that it provides you with the freedom to optimize your code in the future. For example, you might want to prototype a program using Java, to ensure that the concept and design is correct. When you require faster production code, however, you might consider implementing certain performance-intensive methods in native code. The cunning part is that the code that is invoking these methods need not be changed to support the new, optimized native code. As far as Java's concerned, a method is a method and the invocation syntax does not differ.

The Glue Layer

The *glue layer* is the functionality that bridges the gap between Java code and a native method. This layer is required because a Java class, when asked to invoke a native method, only knows a few bits of information about the function it is going to call, such as the name of the function and its parameter types. The class expects the native method *function prototype* to conform to its idea of what the name and parameters are and how they should be declared. The glue layer enforces, via a C include file, the function prototype expected by the Java class.

The C include file that provides the glue layer is generated from a compiled class file. There is exactly one include file for each Java class that contains native methods. You generate the include file using the *javah* program that comes as part of the Sun Java SDK.[2] Here's how to generate the include file for the `HelloWorld` class from Example 4-1:

```
% javah -jni HelloWorld
```

Provided that there are no errors, *javah* creates a file named *HelloWorld.h* in the current working directory. The contents of this header file should look something like what is shown in Example 4-2.

Example 4-2. Header file for HelloWorld.java

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class HelloWorld */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloWorld
 * Method:    printNative
 * Signature: (Ljava/lang/String;)V
 */
JNIEXPORT void JNICALL Java_HelloWorld_printNative
  (JNIEnv *, jobject, jstring);

#ifdef __cplusplus
}
#endif
#endif
```

An include file essentially contains a function prototype for each method you declared as `native` within the Java class. For your native method to be invoked successfully, the implementation of the native method must *exactly* duplicate the function prototype for the native method in this file. Note that the comment for each native method contains the type signature for the method, in case you need it later on.

The `JNIEXPORT` keyword is always present in a native method function prototype before the return type of the native method, to support portable compilation. For example, Unix doesn't require any special declarations to specify that functions are part of a library instead of a program, so `JNIEXPORT` is defined as being empty on Unix systems. Under Win32, however, a function needs to declare itself as `__declspec(dllexport)` to be made part of a library, so `JNIEXPORT` is defined as such on Win32 systems. Similarly, `JNICALL` is defined on Unix platforms as being empty, whereas on Win32 this keyword specifies the type of function calling convention (`__cdecl` or `__stdcall`) that should be used for the method.

The function prototype in the include file also specifies the actual function name for the native method implementation. Every native method name is prefixed with `Java_` to begin with. For a class that does not belong to an explicitly declared package, the class name and method name are then appended to this string, separated by underscores. Thus, in our example, the function name is `Java_HelloWorld_printNative`.

If a class belongs to a package, the fully-qualified class name is appended to `Java_`, with dots replaced by underscores, followed by the method name. If, for instance, our `HelloWorld` class belonged to the `com.oreilly` package, the class name would expand out to `com_oreilly_HelloWorld` and the resulting method name would be `Java_com_oreilly_HelloWorld_printNative`. While this name is not short, it should be unique.

When a class defines multiple native methods with the same name but different parameters (overloaded methods), additional rules are needed to generate unique function names, since the rules we've seen so far would lead to multiple functions with the same name. In this case, the parameters of the method are used to provide differentiation. Whenever a class defines overloaded methods, two underscores are appended to the end of the method name, followed by the type signature of each parameter. There are also some special character sequences that are used to handle Unicode characters, underscores, and array and object parameter types, which include open brackets and semicolons. Table 4-1 shows these rules.

Table 4-1. Special character sequences for parameter type

Type Signature Character	Escape Sequence
A Unicode character *UUUU*	`_0`*UUUU*
The underscore character (`_`)	`_1`
The semicolon in class type signatures (`;`)	`_2`
The open bracket in array type signatures (`[`)	`_3`
The slash character in class type signatures (`/`)	`_`

For example, let's say we have declared the following three native methods within the `HelloWorld` class:

```
public native void printNative();
public native void printNative( String str );
public native void printNative( String str, int indent );
```

These three methods result in the following native method names:

```
Java_com_oreilly_HelloWorld_printNative__
Java_com_oreilly_HelloWorld_printNative__Ljava_lang_String_2
Java_com_oreilly_HelloWorld_printNative__Ljava_lang_String_2I
```

The first native method has no arguments at all, so its name simply has two underscores appended to it—remember that all overloaded native method names separate the method name from the parameters with *two* underscores. The other two methods declare parameters, so the parameter types are appended to the name. Note that the slashes used in the type signature are converted to underscores in the method name, and the semicolon that terminates each class type is converted to `_2`.

The final point you should notice about the function prototype for a native method involves the first two parameters. Every native method declaration includes two parameters that are inserted before the parameters you declared within your Java code. These parameters correspond to the JNI interface pointer for the current executing JVM thread and a pointer to the current object, or `this` in C++ terminology.

As we discussed in the section called *The JNI Execution Environment*, every JNI function is invoked via a JNI interface pointer of type `JNIEnv`. Thus, this parameter is passed to your native method implementation, so you can use it to invoke any JNI functions within your native code and thereby ensure the integrity of your application within the JVM.

The second parameter is equally important, as it preserves the object-oriented nature of Java within your native code, which can be written in a non-object-oriented language, such as C. When a native method is an instance method (i.e., it operates on a particular object), this second parameter is of type `jobject`, as in the case of our `printNative()` method. When a native method is a class (`static`) method, however, the second parameter is of type `jclass`.

Implementing the Native Method Body

Implementing the native method body is, as you might imagine, the most important stage of native method programming. The main work to be done here involves writing the actual code that does whatever your native method needs to do. The complexity of this task obviously depends on what you want your native method to do. Since native methods can be written to do just about anything, there's not much more we can say about this task here.

The one important thing I do want to emphasize is the function prototype for your native method. It is absolutely imperative that you get the function prototype right, or the native method will never be executed.

To go back to our `HelloWorld` class, we know from looking at the *HelloWorld.h* include file that the JVM is expecting the following function prototype for the `printNative()` native method:

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

Now all that remains is to write some native code that prints a message, which the following C++ code does admirably. Notice that the function prototype in Example 4-3 been implemented exactly as specified in the include file.

Example 4-3. Native code body of the native method (nativeHelloWorld.cpp)

```
/**
 * Implements the body of the native method ’printNative()’ declared
 * in the HelloWorld class.
 */

#ifndef WIN32
extern "C" {
#endif /** !WIN32 */

#include <stdio.h>
#include <HelloWorld.h>

JNIEXPORT void JNICALL
Java_HelloWorld_printNative( JNIEnv *env, jobject arg,
                             jstring instring ) {

    const jbyte *str =
        (const jbyte *)env->GetStringUTFChars( instring, JNI_FALSE );

    printf( "Native string: %s\n", str );

    env->ReleaseStringUTFChars( instring, (const char *)str );

    return;
  }

#ifndef WIN32
  }
#endif /** !WIN32 */
```

One final point about your native method implementation is that you need to remember to include the generated include file for each class that has a native method body implemented in the current file. In other words, the following line in *nativeHelloWorld.cpp* is critical:

```
#include <HelloWorld.h>
```

If you forget to include this file, several things can happen, depending on your compiler. The best case scenario is that the compiler generates a warning that the function cannot be referenced or resolved. In the worse case, however, a default function prototype might be inserted in place of the function prototype declared within the include file. If this happens, it is the same as if you have specified the function prototype incorrectly, meaning that your native method will fail upon invocation.

JNI Data Types

Before we go any further with the JNI, we need to take some time to understand the data types used by JNI and how they correlate to Java data types. The JNI data types are the building blocks that all the JNI functions are based on, whether we are talking about values that are being returned from JNI function invocations or parameters that are being passed to JNI functions. Every single piece of data that is passed from Java code to the JNI or created within a native method and passed back to Java is of one of the JNI data types.

Primitive Data Types

For each primitive Java data type, there is a corresponding JNI type, as shown in Table 4-2.

Table 4-2. JNI Primitive Data Type Mappings

Java Type	JNI Type	Size
`boolean`	`jboolean`	unsigned 8 bits
`byte`	`jbyte`	signed 8 bits
`char`	`jchar`	unsigned 16 bits
`short`	`jshort`	signed 16 bits
`int`	`jint`	signed 32 bits
`long`	`jlong`	signed 64 bits
`float`	`jfloat`	32 bits
`double`	`jdouble`	64 bits
`void`	`void`	N/A

As you can see, the mapping between the JNI data types and their Java counterparts is straightforward and intuitive. In addition, the types are declared and manipulated in exactly the same way as the standard Java and C data types. The JNI data types are guaranteed to be the given size, no matter which platform you are writing JNI code for. For example, on an Intel-based microprocessor, an `int` is usually 32 bits in length. However, on a 64-bit microprocessor such as the Digital Alpha, an `int` might be 64 bits. By using `jint` values, you are guaranteed that the values are the same size no matter what platform you are using.

Reference Data Types

The JNI provides reference data types to represent generic objects, classes, strings, arrays, and throwable objects:

`jobject`

This JNI data type corresponds to a generic Java object; objects in JNI are referenced and manipulated via this opaque data type.

In the same way that all objects within Java code can be safely cast to be of type `java.lang.Object`, all reference types within native code can be safely cast to be of type `jobject`. Furthermore, with the exception of the other reference types listed here, there are no specific data types defined for different kinds of objects—all objects are simply of type `jobject`.

As we discussed in the section called *The Glue Layer*, when a native method is an instance method, the second parameter of the function prototype for the native method is always of type `jobject`. This parameter passes the object on which the method was invoked to the native code, so that you can manipulate the object in your code.

`jclass`

This data type corresponds to a Java object of type `java.lang.Class`, which contains information on the attributes of a particular Java class. Whereas a variable of type `jobject` contains a specific instance of a class, a `jclass` variable contains information about the class itself. You use this type of variable when you need to perform operations on a particular class, such as creating new objects of a given class or invoking `static` methods of a class.

JNI provides several functions for locating information on classes. The most commonly used methods are `FindClass()`, which returns a `jclass` value that contains information about the desired class, and `GetObjectClass()`, which returns a `jclass` value that contains information about the class of the given Java object. Other JNI functions, such as `GetFieldID()` and `GetMethodID()`, require a `jclass` argument. This argument is used to retrieve information about the class, such as whether a field or method exists.

`jstring`

This data type corresponds to a `java.lang.String` object within Java (i.e., a string of Unicode characters). Since `String` is a full-fledged class type and almost certainly the most commonly used data type, it is convenient to have a separate data type to represent strings.

The JNI defines several function for manipulating `jstring` variables, including functions to convert strings from the Unicode UTF-8 format to “standard” C/C++ character arrays and back again. There are also functions for calculating the length of a `jstring` and creating a new `jstring` from existing data. These string manipulation functions make it easy to pass strings between Java code and native code.

`jarray`

The `jarray` data type acts as a general data type for arrays. For each primitive data type listed in Table 4-2, there is a corresponding array type, as listed in Table 4-3. `jarray` is the common parent to all of these types, which theoretically means that you can cast arrays of different data types to each other, although this is rarely done.[3]

Table 4-3. JNI Array Data Types

JNI Data Type	Description
`jobjectarray`	Array of `java.lang.Object values`
`jbooleanarray`	Array of `boolean values`
`jbytearray`	Array of `byte values`
`jchararray`	Array of `char values`
`jshortarray`	Array of `short values`
`jintarray`	Array of `int values`
`jlongarray`	Array of `long values`
`jfloatarray`	Array of `float values`
`jdoublearray`	Array of `double values`

The odd-man-out in this list of array data types is `jobjectArray`; it is the only array that doesn’t contain primitive data types. It is, in fact, an array of `jobject` values, which can represent any kind of Java object. Thus, if you have an array of `String` objects (`String[]`) within your Java code as, the corresponding declaration within native code is of type `jobjectArray`.

`jthrowable`

This type corresponds to a `java.lang.Throwable` object, which is a type of object that is thrown when an error occurs. The most common throwable objects are `java.lang.Exception` objects. For example, if you attempt to reference a non-existent field within a given class, an `Exception` is thrown. Your Java code can catch this exception to provide error handling. The JNI provides functions that allow you to implement orthogonal error handling in your native code, as we’ll discuss later in Chapter 11.

Figure 4-2 shows the relationship between the various JNI reference types.

Figure 4-2. JNI Reference Data Types

```
jobject
  ├─ jclass
  ├─ jstring
  ├─ jarray
  │     ├── jobjectArray
  │     ├── jbooleanArray
  │     ├── jbyteArray
  │     ├── jcharArray
  │     ├── jshortArray
  │     ├── jintArray
  │     ├── jlongArray
  │     ├── jfloatArray
  │     └── jdoubleArray
  └─ jthrowable
```

The jvalue Data Type

JNI provides one final data type, `jvalue`, that is defined as a C `union` of all the primitive data types and `jobject`. This data type can be used to construct argument lists to be passed to various JNI functions, when the data type of a given argument is not known in advance. The `jvalue` data type therefore acts opaquely for all the data types it encapsulates. The definition of `jvalue` is as follows in *jni.h*:

```
typedef union jvalue {
    jboolean z;
    jbyte    b;
    jchar    c;
    jshort   s;
    jint     i;
    jlong    j;
    jfloat   f;
    jdouble  d;
    jobject  l;
} jvalue;
```

Notes

1. Or in C++ terminology, a virtual function table.

2. This program supports both the generation of JNI and JDK-1.0 native method glue layers, so you need to specify the `-jni` argument when you want to generate JNI include files
3. For example, you can cast an array of floating-point numbers expressed as a `jdoubleArray` to an array of integers expressed as a `jintArray`, but this causes any fractional parts of the numbers to be truncated.

Chapter 5. Introducing the Raw Native Interface

The two most commonly used Java Virtual Machines are JVMs based on Sun's reference implementation and the Microsoft Virtual Machine. At the time of writing, the Sun JVM is being used by Sun, SGI, Netscape, Symantec, and IBM to name but a few, while Microsoft's JVM is being used solely by Microsoft. This split has an effect on native method programming, since the Sun JVM uses the portable Java Native Interface (JNI), but the Microsoft JVM primarily supports its own proprietary native method interface known as the Raw Native Interface (RNI). The Microsoft JVM now also supports the JNI, but the performance of JNI compared to RNI is not as optimal.

While it is tempting to ignore Microsoft's JVM technology and simply support JNI-based JVMs, Microsoft's JVM is too widely deployed to validate this solution. Internet Explorer, which uses Microsoft's JVM, is now installed as the preferred browser on over 50% of machines. There are also considerable differences in the maturity and performance of JNI-based JVMs, further reducing the efficacy of the JNI path. Finally, the performance of the Microsoft JVM is highly optimized; it has been benchmarked as one of the fastest JVMs available at the time of this writing.

This book covers the RNI alongside the JNI where possible, or in separate chapters where the differences between the two interfaces are difficult to reconcile. This will hopefully give you the information you require to decide whether to support a pure JNI native codebase or a mixed JNI and RNI codebase. Note that even though I am discussing RNI throughout this book, I am not going to cover Microsoft technologies such as J/Direct or COM integration with native methods.

The Microsoft JVM

The Microsoft JVM is a formidable piece of technology that uses extreme internal optimization to provide excellent performance in executing Java code. The Microsoft JVM is quite closely tied to the operating system (at least on Win32 platforms), which has allowed Microsoft to develop code that extracts every last drop of performance out of the JVM. The Microsoft JVM is further help by just-in-time (JIT) compilation.

Given that pure Java code is so fast on a Microsoft JVM, you may be wondering why you would want to use platform- and JVM-dependent native methods? The answer is that native methods are still very useful when it comes to integrating existing legacy code with a Java application. For example, say you have an application that performs financial transactions, with GUIs written for Windows, the Mac, and the X Window System. Rewriting the entire application in Java would mean replicating all the financial transactions functionality. Instead, you might convert the GUI code to Java, but use native methods to access the existing legacy code that performs the real work. This solution has the benefit of reducing development and support on the GUI code, while not losing any of the existing application functionality.

There are fundamental differences in architecture between the JNI-based JVMs and the Microsoft JVM.

As we discussed in Chapter 4, the JNI-based JVMs use a three-tier architecture that abstracts the JNI away from the underlying implementation of each JNI function. This is what affords JNI its portability across JVMs.

On the other hand, the Microsoft JVM directly exposes the internals of the JVM and its data structures to the RNI, giving you far more power over exactly how you want to interface with the JVM. As we'll see as we work with the RNI in later chapters, every Java object stored within the JVM is exposed as a C `struct` instead of as an indirect reference. This provides an extremely direct way of manipulating Java objects and classes from native code. Of course, this power also requires greater responsibility—you must be incredibly careful in choosing the operations you perform and how you perform them, as it is very easy to cause JVM corruption or crashes if you are not careful.

Implementing RNI Native Methods

Just as with the JNI, there are three basic tasks involved with writing native methods using the RNI:

- Declaring the methods you want to implement as native code as `native` methods in the Java class.
- Writing the native code bodies, using the correct function prototypes.
- Generating the "glue" layer that connects the native code and the Java class, to ensure that both pieces know where each other are and can communicate.

Writing The Java Code

Informing the JVM that a method is implemented natively, instead of in Java, is the easiest part of native methods programming. All you do is declare your native methods with the `native` keyword. Example 5-1 shows a Java class that defines a native method that outputs to the screen. The example prints two messages, one using a native method, and the other using Java code to print to `System.out`.

Example 5-1. Native method declaration (HelloWorld.java)

```
/** Invokes a Java-implemented method and a native method */
public class HelloWorld {

    /** static initializer block which loads the native library */
    static {
        try {
            System.loadLibrary( "HelloWorld" );
          } catch ( UnsatisfiedLinkError e ) {
            System.err.println( "Cannot load HelloWorld library: " +
                                e.toString() );
          }
      }

    /** The natively implemented method to print a String */
    public native void printNative( String str );

    /** The Java implemented method to print a String */
    public void printString( String str ) {
        System.err.println( "String printed: " + str );
      }

    /** Create a main() method so that we can run this class from a shell */
    public static void main( String argv[] ) {
        /** Create a new instance of the HelloWorld class */
        HelloWorld hw = new HelloWorld();
      }

    /** Construct a new instance of HelloWorld */
    public HelloWorld() {
        /** Print from Java */
        printString( "This string will get printed from Java!" );

        /** Print from native code */
        printNative( "This string will get printed from native code!" );
      }
  }
```

If you compare Example 5-1 with Example 4-1, you'll see that the two programs are identical. As you'd expect with Java, there's nothing about your Java code that differs between writing native methods for JNI or RNI.

Besides the method that is declared as `native`, this short example demonstrates one other important aspect of native methods programming: the call to `System.loadLibrary()`. You must remember to explicitly load the library that contains your native methods. If you forget to do this, the invocation of any native method will fail and throw an `UnsatisfiedLinkError` in the process.

As you can see from Example 5-1, native methods are invoked in exactly the same way as normal, Java-implemented methods. As we discussed in Chapter 4, this approach has the advantage allowing you to optimize your code in the future, by changing certain performance-intensive methods from Java code to native code. As far as Java's concerned, a method is a method and the invocation syntax does not differ, so all you have to do is change the method declaration to achieve this.

The Glue Layer

As we discussed in Chapter 4, the glue layer is the bridge between Java code and a native method. When a Java class is asked to invoke a native method, it has limited information about the function it is going to call, such as the name of the function and its parameter types. The class expects the native method function prototype to conform to its idea of what the name and parameters are and how they should be declared. The glue layer enforces, via a C include file, the function prototype expected by the Java class.

The C include file that provides the glue layer is generated from a compiled class file. There is exactly one include file for each Java class that contains native methods. Generating an include file is extremely simple and is done using the *msjavah* program that is bundled with the Microsoft Java SDK. Here's how to generate the include file for the `HelloWorld` class from Example 5-1:

```
C:\> msjavah HelloWorld
```

Provided that there are no errors, *msjavah* creates a file named *HelloWorld.h* in the current working directory. The contents of this header file should look something like what is shown in Example 5-2.

Example 5-2. Auto-generated RNI header file (HelloWorld.h)

```
/*  DO NOT EDIT - automatically generated by msjavah  */
#include <native.h>
#pragma warning(disable:4510)
#pragma warning(disable:4512)
#pragma warning(disable:4610)

struct Classjava_lang_String;
#define Hjava_lang_String Classjava_lang_String

/*  Header for class HelloWorld  */

#ifndef _Included_HelloWorld
#define _Included_HelloWorld

#define HHelloWorld ClassHelloWorld
typedef struct ClassHelloWorld {
#include <pshpack4.h>
    const long MSReserved;
#include <poppack.h>
} ClassHelloWorld;

typedef struct ClassArrayOfHelloWorld {
    const int32_t MSReserved;
    const unsigned long length;
    HHelloWorld * const body[1];
} ClassArrayOfHelloWorld;
#define HArrayOfHelloWorld ClassArrayOfHelloWorld
#define ArrayOfHelloWorld ClassArrayOfHelloWorld

#ifdef __cplusplus
extern "C" {
#endif
__declspec(dllexport) void __cdecl HelloWorld_printNative (struct HHel-
loWorld *, struct Hjava_lang_String *);
#ifdef __cplusplus
}
#endif

#endif  /* _Included_HelloWorld */

#pragma warning(default:4510)
#pragma warning(default:4512)
#pragma warning(default:4610)
```

This file might seem a bit confusing at first glance, but the interesting parts are a C `struct` that represents the structure of the class and a function prototype for each method declared as `native` within the class. We'll concentrate on the function prototype for now and talk about the `struct` later in this chapter.

Only methods declared with the `native` keyword have a function prototype within an include file. For your native method to be invoked successfully, the implementation of the native method must *exactly* duplicate the function prototype for the native method in this file.

The `__declspec(dllexport)` keyword is always present in an RNI function protoype. This keyword is required under Win32 when you are compiling a method for use within a library. In order for the function to be visible to other applications that use the library, the function must be exported. To achieve this, a function needs to declare itself as `__declspec(dllexport)`, which is exactly what the include file does for you automatically.

For convenience when writing native method implementations, I tend to replace `__declspec(dllexport)` with `RNIEXPORT` and `__cdecl` with `RNICALL`. I define these as follows in my native method implementations:

```
#define RNIEXPORT __declspec(dllexport)
#define RNICALL __cdecl
```

These definitions make my RNI code easier to type and also causes them to look similar to JNI native methods that are declared using `JNIEXPORT` and `JNICALL`. It also allows you to rapidly alter your code if the definitions generated by *msjavah* change (e.g., if `__cdecl` changes to `__stdcall`). These conventions are used throughout the RNI examples in this book.

The function prototype in the include file also specifies the actual function name for the native method implementation. In order to provide some degree of uniqueness, *msjavah* generates the function name automatically from the class name and the package in which the class is placed. For a class that does not belong to an explicitly declared package, the class name and method name are simply joined together with an underscore (e.g., `HelloWorld_printNative()`).

If a class belongs to a package, the fully-qualified class name is used, with the dot separators replaced by underscores. The method name is then added, separated from the package and class name with another underscore. For example, if our `HelloWorld` example belonged to the `com.oreilly` package, the automatically generated function name for the `printNative()` method would be `com_oreilly_HelloWorld_printNative()`.

Unlike the *javah* program, *msjavah* does not produce different method names when you have overloaded native methods. For example, to specify a point in 2- or 3-dimensional space, a class might contain the following overloaded methods:

```
public native void vertex( int x, int y )
public native void vertex( float x, float y )
public native void vertex( int x, int y, int z )
public native void vertex( float x, float y, float z )
```

Running these method declarations through *msjavah* generates four function prototypes with identical names, but different parameters. Any good C++ compiler should be able to handle this, but compilation with a standard C compiler will probably fail. Thus, if you have overloaded native methods, you should really consider using a C++ compiler.

If you are forced to use a C compiler for some reason, there is a workaround for problem, but it's rather a kludge. The idea is to implement dispatcher methods in Java that simply invoke uniquely named native methods. Here are a couple of `vertex()` methods done using this technique:

```
/* Private native method dispatcher */
private native void vertex2i( int x, int y );

/* Public overloaded method */
public void vertex( int x, int y ) {
    vertex2i( x, y );
  }

/* Private native method dispatcher */
private native void vertex2f( float x, float y );
```

```
/* Public overloaded method */
public void vertex( float x, float y ) {
    vertex2f( x, y );
  }
```

This system preserves both the uniqueness of native method names and the overloaded methods. Of course, there are now two method invocations required instead of one, but at least it works. Obviously, this solution doesn't work very well if you are doing both JNI and RNI versions of your native code.

The final point you should notice about the function prototype for an RNI native method involves the first parameter. Every native method declaration includes a parameter that is inserted before the parameters you declared within your Java code. This parameter corresponds to the current object, which can be correlated with the implicit `this` value in C++ terminology. In our Hello World example, you can see that this parameter is a pointer to a structure of type `HHelloWorld`; we'll learn more about this structure later in the chapter, we when talk about RNI data types. This parameter is important, as it preserves the object-oriented nature of Java within your native code, which can be written in a non-object-oriented language, such as C.

Implementing the Native Method Body

Implementing the native method body is really the heart of native method programming. The foremost issue here is the actual native method content—the actual code that you want to execute when the native method is invoked from Java. The complexity of this code obviously depends on what you want your native method to do, which can be just about anything, so there's not much more I can say about the task here.

The one important thing I do want to emphasize is the function prototype for your native method. It is absolutely imperative that the function prototype is correct or the native method will never execute.

To go back to our `HelloWorld` class, we know from looking at the *HelloWorld.h* include file that the JVM is expecting the following function prototype for the `printNative()` native method:

```
__declspec(dllexport) void __cdecl
    HelloWorld_printNative (struct HHelloWorld *,
                            struct Hjava_lang_String *);
```

Now all that remains is to implement some native code that prints a message, as the following C++ code does admirably. Notice that the function prototype in Example 5-3 been implemented exactly as specified in the include file.

Example 5-3. Native code body of the native method (nativeHelloWorld.cpp>

```
/**
 * Implements the body of the native method ’printNative()’ declared
 * in the HelloWorld class.
 */

#ifndef WIN32
extern "C" {
#endif /** !WIN32 */

#include <stdio.h>
#include "../../rnidefs.h"
#include "HelloWorld.h"

#ifdef RNIVER
DWORD RNIGetCompatibleVersion() {
    return RNIVER;
  }
#endif /** RNIVER */

RNIEXPORT void RNICALL
HelloWorld_printNative( struct HHelloWorld *arg,
                        struct Hjava_lang_String *instring ) {

    /** Temporary buffer */
    char buf[1024];

    /** Extract the Java String */
    javaString2CString( instring, buf, sizeof( buf ) );

    /** Print the string out... */
    printf( "%s\n", buf );
  }

#ifndef WIN32
  }
#endif /** !WIN32 */
```

One final point about your native method implementation is that you must remember to include the generated include file for each class that has a native method body implemented in the current file. In other words, the following line in *nativeHelloWorld.cpp* is critical:

```
#include <HelloWorld.h>
```

If you forget to include this file, several things can happen, depending on your compiler. The best case scenario is that the compiler generates a warning that the function cannot be referenced or resolved. In the worse case, however, a default function prototype might be inserted in place of the function prototype declared within the include file. If this happens, it is the same as if you have specified the function prototype incorrectly, meaning that your native method will fail when it is invoked.

All of the RNI functions are actually defined with an include file called *native.h*. You do not need to explicitly include this file in your code, as all *msjavah*-generated header files automatically include this file for you.

Note that newer versions of the Microsoft Virtual Machine (newer than build 2239 or 2252) as shipped with the Microsoft Java SDK Version 2.0 or higher, Internet Explorer 4 or higher, or separately installed from Microsoft's web site places a new requirement on you regarding RNI programming. When the JVM loads a native library, it attempts to invoke a method called `RNIGetCompatibleVersion()`, which is supposed to return an integer that corresponds to the version number of the RNI implemented in the current JVM. This method is easy to implement; the following stub code is generally all you should ever need to use:

```
#ifdef RNIVER
DWORD RNIGetCompatibleVersion() {
    return RNIVER;
  }
#endif /* RNIVER */
```

`RNIVER` is a symbol defined within *native.h*; it exists in Version 2 only. Thus, the previous stub only attempts to compile the `RNIGetCompatibleVersion()` method if that symbol is defined. The main issue with this approach is that you cannot use a Version 1 compiled library against a Version 2 JVM. There is no such restriction when using a Version 2 compiled library against a Version 1 JVM, however.

RNI Data Types

Before we dive further into the RNI, we need to take a moment to understand the data types used by RNI and how they correlate to Java data types. The RNI data types are the building blocks that all the RNI functions are based on, including values that are returned from RNI function invocations or parameters that are passed to RNI functions. Every single piece of data that is passed from Java code to the RNI or created within a native method and passed back to Java uses one of the RNI data types.

The RNI is far simpler with regard to data types than the JNI. The JNI defines lots of special data types that represent native code reflections of Java objects and data types. RNI, on the other hand, uses simple C data types to represent standard values and C `struct` structures to represent Java classes and objects.

Primitive Data Types

The RNI provides a simple mapping between the primitive Java data types and data types that represent them in native code. This process makes manipulating Java data from within native code far simpler than having to translate values from a format suitable for use within Java to a format suitable for use within native code. Table 5-1 shows the RNI mappings.

Table 5-1. RNI Primitive Data Type Mappings

Java Data Type	RNI Data Type
`boolean`	`long`
`byte`	`long`
`char`	`long`
`short`	`long`
`int`	`long`
`long`	`int64_t`
`float`	`float`
`double`	`double`

As you can see, the mapping between the Java data types and RNI data types is quite obvious and intuitive. All the integer data types are at least 32 bits in length, and the Java `long` type is a 64-bit value. Floating-point data types are preserved as 32-bit `float` values or 64-bit double-precision `double` values. These data types can be used directly in manipulating and setting new values within Java objects or as parameters for method or constructor invocation.

To help you make sure you have mapped a Java data type to the correct RNI data type, each generated include file contains declarations of the fields within that Java class. For example, say we define the following Java class:

```
public class Vertex {

    public float x;
    public float y;
    public float z;

    ...
}
```

If we use *msjavah* to create an include file for this class, we end up with the following code fragment in that include file:

```
#define HVertex ClassVertex
typedef struct ClassVertex {
#include <pshpack4.h>
    const long MSReserved;
    float x;
    float y;
    float z;
#include <poppack.h>
} ClassVertex;
```

As you can see, the `ClassVertex` structure contains declarations that match the fields in the `Vertex` Java class.

Classes and Objects

As we've just seen, when you use *msjavah* on a Java class, the generated include file contains a `struct` definition that mirrors the class declaration. This data structure is the data type that is used to refer to that class in native code. So, with our `Vertex` class, the corresponding RNI data type is `ClassVertex`. In general, the name of the RNI data structure is the fully-qualified name of the Java class, with dot separators replaced by underscores and with `Class` prepended. Thus, if the `Vertex` class were part of the `com.oreilly` package, the name of the data structure would be `Classcom_oreilly_Vertex`.

To refer to an object of a particular class with the RNI, we use a pointer to the data structure defined in the include file. Thus, we can refer to an instance of the `Vertex` class with a `struct HVertex *` variable in native code. By convention, we use a different name for the data structure, however, to indicate that we are dealing with an object. With our `Vertex` example, notice that `HVertex` is defined to be the same as `ClassVertex`. The "H" represents that fact that we are dealing with a "handle" to the object. The general naming convention for referring to objects is to prefix the fully-qualified class name with an `H`. Thus, if the `Vertex` class were part of the `com.oreilly` package, the handle to an object would be declared as `struct Hcom_oreilly_Vertex *`.

The final RNI data type that concerns classes and objects is the `ClassClass` type. A variable of this type contains meta-information about a particular class (i.e., it corresponds to the `java.lang.Class` class in Java). Certain RNI functions that manipulate classes require class information about a particular class. You can get a `ClassClass` variable for a particular Java class using the `FindClass()` or `Object_GetClass()` function, as we'll discuss in more detail in Chapter 9.

The C `struct` defined in the generated include file actually contains fields for all of the fields defined within the Java class and all of its superclasses, right up to the class that inherits from `java.lang.Object`. If you have fields in a subclass that shadow fields in a superclass, the field names in the `struct` are modified slightly to avoid naming collisions. Consider the following Java classes:[1]

```
public class Vertex {

    public float x;
    public float y;
    public float z;

    ...
}

public class Subvertex (

    public float x;

    ....
}
```

Using *msjavah* on `Subvertex` creates an include file that contains the following:

```
#define HSubvertex ClassSubvertex
typedef struct ClassSubvertex {
#include <pshpack4.h>
    const long MSReserved;
    /* Members of Vertex */
    float x_1;
    float y;
    float z;
    /* Members of Subvertex */
    float x;
#include <poppack.h>
} ClassSubvertex;
```

Arrays

Within the RNI, an array is represented by the data type `HArrayOf`*`Type`*, where *`Type`* is one of the primitive data types. For example, an array of `byte` values is represented by the type `HArrayOfByte` in native code. There is also a catch-all array type called `HArrayOfObject`, which represents an array of Java objects. Chapter 7 covers working with arrays in detail.

Notes

1. And ignore why you would ever create pathological classes like this!

Chapter 6. Working with Strings

The string is probably the most commonly used data type in any programming language, so we'll start our discussion of the specifics of native method programming by focusing on strings.

The JNI places a particular emphasis on strings, by providing the `jstring` especially for working with them. Both the JNI and RNI define several functions that allow you to manipulate strings from within native code. These methods can be subdivided into the basic categories of string creation, string conversion, and string sizing. The ability to manipulate strings from within native methods is absolutely essential. An application that uses native methods almost always needs to pass strings into native code and return strings to Java from native code.

The JNI and RNI string manipulation functions are especially important because Java programs are written using the Unicode character set, rather than the ASCII or ISO Latin-1 encoding used by most other programming languages. While Unicode makes internationalization and localization easier, it makes string handling somewhat trickier. The Java `String` class shields you from Unicode, by automatically translating ISO Latin-1 data into Unicode before manipulating the string. Native code does not provide such niceties, however, since it is primarily geared towards an ISO Latin-1 environment. If you reference the data in a Java `String` directly within native code, you'll see the Unicode representation, which appears as complete gibberish unless you have Unicode translation code of your own. Similarly, if you create a string within native code and pass it back into Java, the resulting Java `String` contains rubbish. To avoid needing to write Unicode translation routines yourself, you should use the JNI and RNI functions to handle string data within your native methods.

Creating Strings

The ability to create new strings in your native code is key to being able to return a Java string from a native method. It doesn't take much imagination to see that this is something you'll want to do fairly often.

The JNI defines a function called `NewStringUTF()` that creates a Java `String` object from an array of C/C++ character data. UTF stands for UTF-8, which is an ISO Latin-1 (and ASCII) compatible transformation format for Unicode characters. The function actually returns a `jstring` value, which is equivalent to a `String` when it gets passed back to Java code. In our symbol stones example, the name of the stone can be queried by invoking the `getName()` method against a `SymbolStone` object. This native method returns a Java `String` and is written as follows:

```
/** Returns the name of the stone */
JNIEXPORT jstring JNICALL
Java_SymbolStone_getName( JNIEnv *env, jobject arg ) {
    /** Extract the pointer */
    jfieldID pDataFieldID = \
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" ); \
    SymbolStone_t *stone = \
        (SymbolStone_t *)env->GetIntField( arg, pDataFieldID );

    /** Safety check */
    if ( stone == NULL || (int)stone == -1 ) {
        return NULL;
      }

    /** Create a new Java String from the name */
    return env->NewStringUTF( stone->name );
  }
```

`NewStringUTF()` is designed only for creating strings from 8-bit ISO Latin-1 data. If the character data is your native code is 16-bit data, you should use the JNI `NewString()` function instead.

Creating Java strings using the RNI is equally straightforward; the function that performs this task is called `makeJavaString()`. This function simply takes a C string and its length as arguments and returns the new Java `String` (`Hjava_lang_String`) if it is created successfully. For example, with the RNI, the `World.getName()` method can be written as follows:

```
/** Returns the name of the stone */
RNIEXPORT struct Hjava_lang_String * RNICALL
SymbolStone_getName( HSymbolStone *arg ) {
    /** Extract the pointer */
    SymbolStone_t *stone = (SymbolStone_t *)arg->_pData;

    /** Safety check */
    if ( stone == NULL || (int)stone == -1 ) {
        return NULL;
      }

    /** Create a new Java String from the name */
    return makeJavaString( stone->name, strlen( stone->name ) );
  }
```

The RNI function `makeJavaStringFromUtf8()` is similar to `makeJavaString()`, except that it takes a C string that contains null-terminated Unicode UTF-8 formatted data, instead of ISO Latin-1 data. One final RNI function, `makeJavaStringW()`, converts non-UTF-8 Unicode character data to a Java `String`.

Converting Strings

By converting strings, I mean the ability to convert the Unicode data stored within Java string objects to a form that can be used in native code. The `SymbolStone` class declares a method called `load()` which takes a string argument that specifies the name of a file containing symbol stone data.

Thus, we need to convert the filename passed into the native method from Unicode to ISO Latin-1. Both the JNI and RNI define functions that allow you to convert Java `String` objects into an array of characters.

Conversion with JNI

The JNI function for converting a Java `String` object into a character array is called `GetStringUTFChars()`. This function extracts the ISO Latin-1 representation of a Unicode string into an array of characters that can be manipulated in the same way as any other C/C++ array.

`GetStringUTFChars()` makes life simple for you by working out the size of the string being converted and allocating the memory needed to store it. However, just because the function does the memory allocation for you doesn't mean that you don't have to worry about memory management—you still need to remember to deallocate the buffer before you return from the native method. If you return to Java without performing the deallocation, a memory leak will occur. If you are converting Java strings on a regular basis, this memory leak can affect the performance and stability of your program. To deallocate the storage created by `GetStringUTFChars()`, call `ReleaseStringUTFChars()` with the original Java string object and the new character buffer as arguments.

`GetStringUTFChars()` can actually operate in two different ways, depending on the garbage collection state and the JVM upon which your code is executing, among other things. One mode involves `GetStringUTFChars()` returning a direct pointer to the converted string data—in other words, no additional memory is allocated and no string data is transferred to a buffer. This can happen when the Java `String` object's data is already 8-bit data and is stored contiguously within the JVM. Otherwise, `GetStringUTFChars()` allocates a temporary buffer and copies the Java `String` data to it. This is far slower and uses more temporary memory while the string data is being copied about.

If you want to know which technique `GetStringUTFChars()` has used, you can pass the address of a `jboolean` variable as the second argument to the function. When the function returns, the variable contains either `JNI_TRUE`, to signify that a new buffer has been allocated, or `JNI_FALSE`, to indicate that you are operating directly on the internal `String` data. Regardless of the technique the JVM uses, I recommend that you always call `ReleaseStringUTFChars()` when you are done with the string, to avoid any possibility of memory leaks. This function does the right thing no matter whether a buffer is allocated or not.

With all that explanation out of the way, let's look at how to use `GetStringUTFChars()`. Here is the JNI native method implementation of the `load()` method:

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
JNIEXPORT jobjectArray JNICALL
Java_SymbolStone_load( JNIEnv *env, jclass arg, jstring filename ) {

    /** The return array of objects... */
    jobjectArray rv = NULL;

    /** Extract the filename */
    char *filenameChars =
        (char *)env->GetStringUTFChars( filename, NULL );

    /** Load the data from file into C structures... */
    int numRecords = 0;
    SymbolStone_t *stones = load( filenameChars, &numRecords );
    SymbolStone_t *sptr = stones;

    /** Release the filename */
    env->ReleaseStringUTFChars( filename, filenameChars );

    ...

    return rv;
  }
```

I’ve used a coding convention in the previous example that I’ll continue to use throughout this book. When extracting the data from a string, I store the extracted data in a variable that uses the name of the string with “Chars” appended (e.g., `filenameChars`). This makes the code easier to read and also allows you to track down cases where you might have forgotten to release the buffer.

In the previous implementation, we didn’t worry about whether `GetStringUTFChars()` made a copy of the string. If we care to know, however, we can alter the code in the following way:

```
    ...

    /** Stores our copy status */
    jboolean isCopy;

    /** Extract the filename */
    char *filenameChars =
        (char *)env->GetStringUTFChars( filename, &isCopy );
    if ( isCopy == JNI_TRUE ) {
        fprintf( stderr, "-> A copy has been made of the String data\n" );
      } else {
        fprintf( stderr, "-> No copy of the String data has been made\n" );
      }

    ...
```

The `GetStringUTFChars()` and `ReleaseStringUTFChars()` functions both handle only 8-bit character data. If your native code uses 16-bit character data, say to represent Chinese or Japanese characters, you need to use functions that preserve the “wide” Unicode characters. These Unicode-aware functions are `GetStringChars()` and `ReleaseStringChars()`. Since the names of the 8-bit and 16-bit functions are so similar, you need to be careful not to mix them up or your application may start producing strange string data.

As of Java 2, Version 1.2, the JNI adds several new functions for working with strings. For example, there are two new functions that allow you to extract a substring directly from a Java `String`: `GetStringRegion`, which performs the substring operation on a Unicode string, and `GetStringUTFRegion()`, which performs the substring operation on a UTF-8 representation of the string. These functions take parameters that specify the starting index for the substring, the number of characters in the substring, and a pointer to a programmer-allocated buffer into which the substring data is copied.

For example, when you are working with the filename that specifies the data file to load symbol stone data from, you might want to split the filename into two sections, an eight-character prefix and a three-character suffix that conform to DOS standards. Here is a partial implementation of the `SymbolStone.load()` native method that does this using `GetStringUTFRegion()`:

```
    ...

    /* Allocate the buffers for prefix and suffix for filename splitting */
    jchar *prefix = (jchar *)malloc( sizeof( jchar ) * 8 );
    jchar *suffix = (jchar *)malloc( sizeof( jchar ) * 3 );

    /* Extract the regions of the filename */
    env->GetStringUTFRegion( filename, 0, 8, prefix );
    env->GetStringUTFRegion( filename, 9, 3, suffix );

    ...

    /* Release the region buffers */
    free( prefix );
    free( suffix );
    ...
```

Since this code allocates buffers for the substrings with `malloc()`, it deallocates them with `free()`. Whenever you use `GetStringRegion()` or `GetStringUTFRegion()`, you must remember to handle both the allocation and, more importantly, the deallocation within your application.

Another new function as of Java 2, Version 1.2, is `GetStringCritical()`. This function allows you to extract a string and then rely on the fact that the JVM will not move the extracted string data. Using this function also makes it more likely that the extracted string data is a direct pointer to the `String`, not a copy. In other words, when you use `GetStringCritical()`, the JVM may internally reorganize the string data so that it can provide a direct pointer.

`GetStringCritical()` can be used interchangeably with `GetStringChars()` (it returns `jchar` data), although there are some restrictions on its use. When you call `GetStringCritical()`, the JVM assumes that it is entering a *critical section* of code that cannot be interfered with. As such, you must not call any other JNI functions except the corollary `ReleaseStringCritical()`, which releases the extracted string data. `ReleaseStringCritical()` marks the end of the critical section, so the JVM can assume normal functioning at that point. Inside the critical section, you must also be sure that you don't cause the thread in which the native method is being invoked to block, or the JVM's internal state may become unstable.

The `GetStringCritical()` and `ReleaseStringCritical()` functions should be used with the utmost of care. They should only be used when you absolutely require direct access to the `String` data and the JVM would not ordinarily provide this access. Despite these warnings, you should know that these functions do provide a far faster access path to your string data than `GetStringChars()`, so they are definitely worth using in the appropriate circumstances.

Conversion with RNI

The RNI is far simpler than the JNI with regards to converting Java `String` objects. The RNI defines a single method, `javaString2CString()`, to extract the string data convert it from Unicode UTF-8 characters into ISO Latin-1 characters. The resulting string is stored in a buffer that has been pre-allocated by the programmer.

The following code shows the section of the `SymbolStone.load()` and illustrates how we can convert a Java string to a standard C string.

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
RNIEXPORT HArrayOfObject * RNICALL
```

```
SymbolStone_load( ClassSymbolStone *arg,
                  struct Hjava_lang_String *filename ) {

    /** The return array of objects... */
    HArrayOfObject *rv = NULL;

    /** Temporary buffers */
    char filenameChars[1024];

    /** Extract the filename */
    javaString2CString( filename, filenameChars, sizeof( filenameChars ) );

    /** Load the data from file into C structures... */
    int numRecords = 0;
    SymbolStone_t *stones = load( filenameChars, &numRecords );
    SymbolStone_t *sptr = stones;

    ...
```

Sizing Strings

The final aspect of string manipulation involves the ability to calculate the length of a string. If you have already extracted an ISO Latin-1 representation of a Java `String`, you can simply use the C/C++ `strlen()` function to get the length of the string. There are times, however, when you might want to calculate the length of a Java `String` without having to extract it first.

For Java `String` objects that contain data that is representable in the ISO Latin-1 character set, you can call the JNI function `GetStringUTFLength()` to get the length of the data containedin the `String`. If your Java `String` contains Unicode data, use the JNI function `GetStringLength()` instead. Obviously, if you mix these functions up, the sizing information for the string will probably be wrong.

For example, we might need to verify that the filename for the symbol stone data file is less than or equal to 14 characters in length, in case we have to deploy the software on older UNIX flavors. Thus, when we call the `SymbolStone.load()` method, we use `GetStringUTFLength()` to check that the filename isn't too long.

```
    ...

    /*
     * Check that the filename length <= 14 chars. Return NULL if it's too
     * long
     */
    if ( env->GetStringUTFLength( filename ) >= 14 ) {
        return NULL;
      }

    /** Otherwise, load up the data file... */
    ...
```

If you are using RNI, you can use the function `javaStringLength()` to retrieve the length of a Java `String`. This function takes into account the different representations of strings (i.e., 16-bit Unicode values or simple UTF-8 values). The following code shows the RNI implementation of the `load()` method:

```
    ...

    /* Check that the filename is <= 14 characters long */
    if ( javaStringLength( filename ) >= 14 ) {
        return NULL;
      }

    /** Otherwise, load the data file... */
    ...
```

Chapter 7. Manipulating Arrays

Arrays are quite commonly used within Java code. Within native code, arrays can be used to provide string functionality and to shield programmers from explicit pointer manipulation. As such, it is quite probable that at some point your Java code and your native code will need to share array data. Fortunately, the JNI and RNI provide functionality to help you work with arrays. These functions support array creation, array manipulation, and array sizing.

One of the issues of working with arrays involves the organization of array data in memory. For instance, if you declare an array of `int` values within native code, the data is guaranteed to inhabit a contiguous segment of memory. In other words, with a pointer to the array, you can advance sequentially through memory and see your integer values. The JVM, however, does not necessarily store array data sequentially—in particular, a JVM may have special ways of storing array data to support optimal garbage collection. Fortunately, the JNI and RNI array manipulation functions coalesce the array elements into a contiguous buffer, allowing you to treat them as a normal C/C++ array.

Creating Arrays

The ability to create new Java arrays from within your native code is absolutely essential. One common use involves legacy code that returns an array, that you then need to pass back to Java. Another possble use arises when you are performing massive amounts of array manipulations, such as with a huge spreadsheet. Array manipulations in Java can be considerably slower than in native code, in part due to the non-contiguous nature of Java arrays. Thus, in an array-intensive Java application, it may be faster to drop into native code, make a native copy of the Java array data, perform the calculations on the native array, and pass the result back.

The basic techniques for creating arrays with the JNI and RNI are quite similar, but the JNI provides far more functions defined for creating arrays and has a more abstracted design than the RNI, so we'll consider each API separately.

Creating Arrays with JNI

The JNI provides a battery of functions for array creation, but these functions are all virtually identical and differ only in the data type of the new array. These functions can be further split into functions that create arrays of Java objects or arrays of primitive data types. And to make life really simple, the JNI functions all have names of the form `New`*`Type`*`Array`, where *`Type`* is the type of the array to create. With all of the primitive type functions, all you have to specify is the number of elements in the array being created.

The JNI array creation functions operate as though you were creating an array within Java. For example, consider the following Java code:

```
int[] array = new int[5];
```

This code creates an array of integers, where each element is initialized to a default value (zero). Similarly, if you create an array of objects, such as `String` objects, the elements are initialized to the default value of `NULL`. To do anything meaningful with the array, you need to set the value of each element to a particular value.

This functionality is reflected in the JNI functions for array creation. When you use a JNI function to create a primitive array, each array element is initially set to zero. When you create an array of Java objects with `NewObjectArray()`, however, you must specify an object that represents the default value for each elemust. If you don't want to initialize the elements of an object array to a default value, you can pass `NULL` as the value of the initial element, to create a placeholder array. In this case, though, you must remember that you cannot access any of the elements of the array until you manually create them or a `NullPointerException` is thrown. If you do specify a default object, you need to be aware that each element is a reference to the default object you have specified. In other words, if you make a change to the object referenced by a particular array element, all of the array elements are similarly altered. To make each element reference a unique object, you have to manually create and set each array element after you create the array.

To illustrate the use of these functions, let's return to our symbol stones example. Once we have created a `SymbolStone` object, we can query its dimensions indirectly via a `Dimension3D` object. The `getDimensions()` method declared within this class return the width, height and depth as a 3 element array of floating point numbers. The implementation of this native method follows:

```
/**
 * Returns a 3 element float array containing the width, height and depth
 */
JNIEXPORT jfloatArray JNICALL
Java_Dimension3D_getDimensions( JNIEnv *env, jobject arg ) {

    /** Extract the pointer */
    jfieldID pDataFieldID =
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" );
    Dimension3D_t *dim =
        (Dimension3D_t *)env->GetIntField( arg, pDataFieldID );

    /** The return array of objects... */
    jfloatArray rv = NULL;

    /** Safety check */
    if ( dim == NULL || (int)dim == -1 ) {
        return NULL;
      }

    /** Allocate the array */
    rv = env->NewFloatArray( 3 );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array\n" );
        return NULL;
      }

    ...

    return rv;
  }
```

Note that this example does not show the code that populates the new array. I've omitted this code for now because it uses JNI functions for accessing array elements that we haven't discussed yet. We'll see a complete implementation of this method later in the chapter.

Using `NewObjectArray()` to create an array of Java objects is slightly more involved than creating an array of a primitive data type. With `NewObjectArray()`, you have to supply not only the number of elements in the array, but also the class type for the array elements (as a `jclass`) and a default object of that type. As we already discussed, the default object can be `NULL`.

To illustrate this, we will look again at the `SymbolStone.load()` method which creates and returns an array of `SymbolStone` objects from a data file. The relevant part of this method is implemented as follows:

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
JNIEXPORT jobjectArray JNICALL
Java_SymbolStone_load( JNIEnv *env, jclass arg, jstring filename ) {

    /** The return array of objects... */
    jobjectArray rv = NULL;

    /** Load the data from file into C structures... */
    ...

    /** Allocate an array of jobjects and initialize to NULL... */
    rv = env->NewObjectArray( numRecords, arg, NULL );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    /**
     * Iterate through each record, allocate a new object and store
     * in the array
     */
    ...

    return rv;
  }
```

As with the `Dimension3D.getDimensions()` implementation, this example only shows the array creation. It omits the code that populates the array with useful values, as that code uses functions we haven't discussed yet. We won't actually see a full implementation until Chapter 8, since we need to discuss object creation before we can implement the appropriate functionality.

Creating Arrays with RNI

As we saw in Chapter 5, a Java array is represented in RNI native code by the data type `HArrayOf`*`Type`*, where *`Type`* is one of the primitive data types. For example, an array of `int` values is represented by the type `HArrayOfInt` in native code. There is also a catch-all array type called `HArrayOfObject`, which represents an array of Java objects. All of these array types share a common way of referencing and manipulating the information stored within them.

The RNI defines two functions for creating arrays: one for creating arrays of primitive data types (e.g., `int`, `float`) and one for creating arrays of Java objects (e.g., `SymbolStone`). The `ArrayAlloc()` function should be used when you want to allocate an array of a primitive data type. This function returns a pointer to an array of the requested type, if successful. `ArrayAlloc()` takes a type argument of the form `T_`*`Type`* (e.g., `T_FLOAT`, `T_CHAR`). You also have to specify the number of elements in the array.

Here's an RNI implementation of the `Dimension3D.getDimensions()` native method that shows the use of `ArrayAlloc()`:

```
/**
 * Returns a 3 element float array containing the width, height and depth
 */
```

```
RNIEXPORT HArrayOfFloat * RNICALL
Dimension3D_getDimensions( HDimension3D *arg ) {

    /** Extract the pointer */
    Dimension3D_t *dim = (Dimension3D_t *)arg->_pData;

    /** The return array of objects... */
    HArrayOfFloat *rv = NULL;

    /** Safety check */
    if ( dim == NULL || (int)dim == -1 ) {
        return NULL;
      }

    /** Allocate the array */
    rv = (HArrayOfFloat *)ArrayAlloc( T_FLOAT, 3 );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array\n" );
        return NULL;
      }

    ...

    return rv;
  }
```

This example does not show the code that copies the native float array into the Java array. I've omitted this code for now because it uses RNI functions that we haven't discussed yet. We'll see a complete implementation of this method later in the chapter.

The other array allocation function defined within the RNI is `ClassArrayAlloc()`. This function is almost identical in operation to `ArrayAlloc()`, except that it allocates an array of objects of a particular class. With `ClassArrayAlloc()`, you specify a type argument of `T_CLASS`, the number of elements in the array, and an additional class name that indicates the class type of the array elements.[1]

The `SymbolStone.load()` native method implementation uses this function to allocate an empty array of objects that will be populated as records are read from the data file.

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
RNIEXPORT HArrayOfObject * RNICALL
SymbolStone_load( ClassSymbolStone *arg,
                  struct Hjava_lang_String *filename ) {

    /** The return array of objects... */
    HArrayOfObject *rv = NULL;

    /** Load the records from the data file */

    /** Allocate an array of objects... */
    rv = (HArrayOfObject *)ClassArrayAlloc( T_CLASS, numRecords, "Symbol-
Stone" );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    /**
     * Iterate through each record, allocate a new object and store
     * in the array
     */
    ...

    return rv;
  }
```

As with the other array creation examples, the code I've shown here only creates the array of `SymbolStone` objects—it does not populate the array. We'll look at the code that populates the array in Chapter 9, after we've learned about object creation. The important point to remember is that if you simply call `ClassArrayAlloc()` to create an array and then return that array, accessing any element results in a `NullPointerException`, since all the array elements are `NULL`. This behavior mirrors that of creating arrays of objects within Java code.

Newer versions of the Microsoft JVM have added a new RNI function, `ClassArrayAlloc2()`, that has the same behavior as `ClassArrayAlloc()` with one slight difference. Instead of taking a string that specifies the class name as its third argument, it takes a pointer to a class information structure of type `ClassClass`. This structure is discussed in detail in Chapter 9.

Manipulating Array Elements with JNI

Within Java, simply creating an array does not initialize the elements of that array to meaningful values. As we've seen, the same is true when you are creating a Java array with JNI. In order to set the array elements to meaningful values, we need to be able to access the individual elements with JNI functions. There are also other situations when we need to be able to manipulate array elements in native code, such as when a native method takes an array argument and we need to work with the individual elements of that array. The techniques for accessing and setting array elements is different between primitive type arrays and object type arrays, so we'll discuss the two types separately.

Manipulating Primitive Array Elements

As with array creation, JNI provides a group of functions for accessing primitive array elements. These functions extract the elements of the given Java array into a chunk of contiguous memory that acts exactly like an array definition in C or C++. You can then traverse the elements of the array in an extremely simple and efficient manner within your native code. Once you have finished manipulating the elements, you can copy those changes back into the original Java array and free up any memory used in extracting the array elements.

The JNI array extraction functions take the form of `Get`*`Type`*`ArrayElements()`, where *`Type`* is the data type of the array elements. Each function returns a prointer to a section of contiguous memory that is created and filled with the elements of the Java array. You can use this pointer to carry out array and pointer operations in your native code. Once you have finished manipulating the extracted elements of the array, you can propagate those changes back into the original Java array using the corresponding `Release`*`Type`*`ArrayElements()` function. Each of these functions copies any changes in the extracted array element data back into the Java array, if desired, and also deallocates the temporary buffer into which the extracted elements were stored. It is extremely important that you remember to release the extracted array elements—if you don't, serious memory leaks can occur.

Now that we know how to manipulate primitive array elements, we can return to the `Dimension3D.getDimensions()` method and look at the complete implementation. This method creates a new array of `float` values, extracts the array elements so that we can insert the position values from the native library, and then sets the new values back to the Java array:

```
/**
 * Returns a 3 element float array containing the width, height and depth
 */
JNIEXPORT jfloatArray JNICALL
Java_Dimension3D_getDimensions( JNIEnv *env, jobject arg ) {

    /** Extract the pointer */
    jfieldID pDataFieldID = \
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" ); \
    Dimension3D_t *dim = \
        (Dimension3D_t *)env->GetIntField( arg, pDataFieldID );

    /** The return array of objects... */
    jfloatArray rv = NULL;

    /** Safety check */
    if ( dim == NULL || (int)dim == -1 ) {
        return NULL;
      }

    /** Allocate the array */
    rv = env->NewFloatArray( 3 );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array\n" );
        return NULL;
      }

    jfloat *rvElements =
        (jfloat *)env->GetFloatArrayElements( rv, NULL );
    rvElements[0] = dim->width;
    rvElements[1] = dim->height;
    rvElements[2] = dim->depth;

    env->ReleaseFloatArrayElements( rv, rvElements, 0 );

    return rv;
  }
```

I follow a personal naming convention for variables that contain extracted elements of an array: the name of the array with `Elements` appended. This convention makes it easier to keep track of which elements are associated with which arrays. It also helps locate situations where you may have forgotten to call `Release`*`Type`*`ArrayElements()` on an array.

Depending on the way a particular JVM organizes array element data internally, the `Get`*`Type`*`ArrayElements()` routines can either allocate a new memory buffer and copy the array elements into the buffer or give you a pointer to the actual array data stored within the JVM itself. This is important for several reasons. If the returned pointer is for a copy of the original array elements, then if you make any changes to those elements and return to Java, the changes you have made will be lost. However, if the returned pointer points at the original Java array data, you must be very careful not to make any mistakes in the operations that you perform on the data, in case you invalidate the data contained within the Java array. Each method of operation has pros and cons: getting a copy of the Java array data provides slower access but it is safer to modify the elements, while a direct pointer to the elements provides faster access but it is more more dangerous to modify the elements.

Unfortunately, the JNI does not allow you to specify which mode you want these functions to use—it simply tells you whether or not a copy has been made. Of course, this knowledge is still useful. Knowing that the JNI is making a copy of the array data allows you to decide how often to call `Get`*`Type`*`ArrayElements()`. For example, the cost of this method in copying a large array may overshadow the performance savings that using native code provides, so you may decide not to use native code.

The final parameter of `Get`*`Type`*`ArrayElements()` can be `NULL`, to specify that you don't want to know whether the extracted elements are a copy or a direct reference to the array contents. If you do want to know, however, simply specify the address of a `jboolean` variable. The variable is set to `JNI_TRUE` if the array elements are a copy or `JNI_FALSE` if the function returns a direct reference. For example:

```
jboolean isCopy;
jfloat *arrayElements =
    env->GetFloatArrayElements( array, &isCopy );

if ( isCopy == JNI_TRUE ) {
    printf( "A copy was made of the elements!\n" );
  } else {
    printf( "No copy was made of the elements!\n" );
  }
```

Critical sections

As of Java 2 Version 1.2, the JNI provides two new methods for array element extraction: `GetPrimitiveArrayCritical()` and `ReleasePrimitiveArrayCritical()`. When the JVM does not support the concept of *array pinning*, the elements of an array that have been extracted in native code may be moved, resulting in wrong results when accessing the array elements. (Pinning is discussed in more detail in Chapter 11, when we talk about global and local references.) Consider the following scenario: you access the array elements using `Get`*`Type`*`ArrayElements()`, which returns a direct pointer to the array elements; garbage collection occurs, moving the array and its contents; you attempt to access the elements via the pointer. At best, you get incorrect results, and, at worst, your application crashes.

`GetPrimitiveArrayCritical()` enforces that the array elements and the array itself are not subject to garbage collection. When you call this function, the JNI considers the function invocation as marking the beginning of a *critical section*, a section of code in which the array is safe from the machinations of the garbage collector. The critical section is ended as soon as `ReleasePrimitiveArrayCritical()` is called.

`GetPrimitiveArrayCritical()` is similar to the normal array element extraction methods, but works for all array types. One of the parameters is declared as a `jarray`, the supertype of all JNI arrays, and the return type is `void *`, which is essentially an opaque pointer that can be freely cast to whichever type is expected. For example, here is how to create a safe set of array elements for a `float` array:

```
/* An array of floating point values created elsewhere */
jfloatArray array;

/* Are we making a copy of the array elements? */
jboolean isCopy;

/* The elements of the floating point array */
jfloat *arrayElements =
    (jfloat *)env->GetPrimitiveArrayCritical( array, &isCopy );
```

In addition to preventing garbage collection, `GetPrimitiveArrayCritical()` makes a better attempt to return a direct pointer to the internal array data than the standard array element extraction functions. Direct access to the array data is not guaranteed, however, so you should use the final parameter to `GetPrimitiveArrayCritical()` to check whether copying has occurred. The potential performance benefits from direct access are substantial, so I'd recommend using `GetPrimitiveArrayCritical()` for any serious array manipulation in your native code.

You must not make any other JNI calls between calling `GetPrimitiveArrayCritical()` to extract the array elements and releasing them with `ReleasePrimitiveArrayCritical()`. Doing so can cause internal instability in the JVM. Note, however, that you can nest `GetPrimitiveArrayCritical()` and `ReleasePrimitiveArrayCritical()` calls, if more than one array needs to be manipulated within a critical section simultaneously. The only additional restriction here is that you release the arrays in the reverse order in which they were acquired. For example:

```
jfloat *arrayOneElements =
    (jfloat *)env->GetPrimitiveArrayCritical( arrayOne, NULL );
jfloat *arrayTwoElements =
    (jfloat *)env->GetPrimitiveArrayCritical( arrayTwo, NULL );

env->ReleasePrimitiveArrayCritical( arrayTwo, arrayTwoElements, 0 );
env->ReleasePrimitiveArrayCritical( arrayOne, arrayOneElements, 0 );
```

Committing changes

As we've already discussed, when you use one of the `Get`*`Type`*`ArrayElements()` functions, the JNI may allocate memory to store the extracted array elements, so you have to deallocate the memory before you return to Java code to avoid memory leaks. And if you've altered any array elements, you probably want to copy those changes back to the Java array. You can accomplish both of these tasks with the `Release`*`Type`*`ArrayElements()` functions, as we've already seen.

What we haven't discussed is that the `Release`*`Type`*`ArrayElements()` functions have a few different modes of operation, depending on how you want to handle any changes you've made to the array elements. The final parameter of the `Release`*`Type`*`ArrayElements()` functions controls the mode that is used. The valid values are: `0` (zero), `JNI_COMMIT`, and `JNI_ABORT`.

The most basic mode is the one that we've already seen: copying the altered data back into the Java array and releasing any allocated memory. To do this, simply call `Release`*`Type`*`ArrayElements()` with a final argument of `0`. This operating mode copies the array data back into the Java array and releases any memory used to store the temporary array elements.

If you don't want to copy the altered array elements back into the original Java array, make the final argument `JNI_ABORT`. This can be useful when you are simply accessing the elements of a Java array within native code, but not actually altering them. For example, your legacy code might have a function that is passed an array of `float` values that you currently have within Java. The legacy code doesn't alter the array elements at all, so it is expensive and unnecessary to copy all the array elements back into Java, When you pass `JNI_ABORT` as the final argument to `Release`*`Type`*`ArrayElements()`, the function deallocates any memory used by the extracted array elements, but does *not* copy the elements back into the Java array.

The final mode of operation, which is not very commonly used but can be extremely useful, emulates a database-style commit procedure. When lots of changes are being made to an array within native code and some other process or thread is waiting for the updated data, you can use this mode to makes sections of updated data available before the entire array has been processed. For example, say you are performing some set of massive stock market calculations that required real-time updates of stock prices stored in an array. In this case, waiting for the calculations on all the companies to complete would result in decidedly non-real-time data. To make the system more dynamic, you might update the Java array after each company calculation is complete. However, since other companies' information is still waiting to be calculated, you don't want the memory containing the extracted array elements to be released—you simply want the array values to be copied. Passing `JNI_COMMIT` as the final argument to a `Release`*`Type`*`ArrayElements()` call provides this mode of operation. Of course, once you have finally finished with the array elements, you need to ensure that you release the memory used by the extracted array elements by calling `Release`*`Type`*`ArrayElements()` with a different mode value.

Using `JNI_COMMIT` to transfer updated data back into Java on a regular basis and on large arrays can be extremely expensive. To provide faster data copying and higher overall application performance, you should consider using the `Set`*`Type`*`ArrayRegion()` functions instead, as we'll discuss in the next section.

As a final note on copying array elements back to Java, let me remind you that the `Release`*`Type`*`ArrayElements()` functions are only relevant for copying data back to Java when the extracted elements are a *copy* of the original data. If you are dealing with a direct pointer to the Java data, any changes you make happen in-place and are visible immediately.

Ranges

You can also extract a range of elements from an array, instead of extracing the entire array. The JNI again defines a function for each primitive data type, in the form of `Get`*`Type`*`ArrayRegion()`. These functions are most useful when you only want to manipulate or reference a small portion of a large array, to avoid the performance and memory overhead of extracting all of the array elements.

One important way in which the `Get`*`Type`*`ArrayRegion()` functions differ from the `Get`*`Type`*`ArrayElements()` functions is that the memory buffer to which the region of elements is extracted is *not* allocated by the JNI. With the `Get`*`Type`*`ArrayRegion()` functions, you must allocate this buffer prior to invocation. This preallocated buffer should be the same size as the number of elements that you want to extract from the array and it should be of the correct JNI data type. For example, to extract a region of 100 integers from a Java array, you need to create a buffer of type `jint` within your native code that can contain 100 `jint` values.

When you use `Get`*`Type`*`ArrayRegion()` to extract a region of elements and manipulate them in some way, you can use `Set`*`Type`*`ArrayRegion()` to copy the changes back to the Java array. Here is a code fragment that shows the use of these functions:

```
/* Start element of the range and size of the batches */
int startElement = 0;
int batchSize = 100;

/* The array created elsewhere */
jintArray array;

/* The length of the array */
int arrayLength = env->GetArrayLength( array );

/*
 * Allocate the buffer to store 100 jint values to be extracted
 * from the array
 */
jint *arrayElements = (jint *)malloc( sizeof( jint ) * 100 );

/* Extract the array elements from the array in batches */
while ( startElement < arrayLength - 1 ) {
    int thisBatchSize =
        ( startElement + arraySize < arrayLength ) ?
          startElement + arraySize : arrayLength - 1;
    for ( int i = startElement ; i < thisBatchSize ; i++ ) {
        /* Extract the array region */
        env->GetIntArrayRegion( array, startElement, thisBatchSize,
                                arrayElements );

        /* Manipulate the array elements somehow */
        ...

        /* Update the results back into the Java array */
        env->SetIntArrayRegion( array, startElement, thisBatchSize,
                                arrayElements );
      }
    startElement += thisBatchSize;
  }

/* Clean up */
free( arrayElements );
```

This example is reasonably optimal as it reuses the same native `jint` buffer for each iteration of the loop, as opposed to allocating and deallocating a new buffer for each iteration. Note also that the example frees the `arrayElements` buffer when the array manipulations are complete. You need to remember to do this to avoid memory leaks in your native code.

`Set`*`Type`*`ArrayRegion()` can also be used to copy back a portion of an array, even if you have extracted the entire contents using `Get`*`Type`*`ArrayElements()`. calls. This is an excellent way to rapidly update a massive array in a real-time way. The short stock market scenario outlined earlier would benefit greatly from using this technique. The only restriction on using `Set`*`Type`*`ArrayRegion()` is that the region you want to copy must be contiguous in the array, although it can of course be as small as a single element. Finally, if you do use `Set`*`Type`*`ArrayRegion()` to copy portions of an array extracted with `Get`*`Type`*`ArrayElements()`, you still need to remember to call `Release`*`Type`*`ArrayElements()` to release any allocated memory, as well as to explicitly free any buffers you created to work with the array regions.

Manipulating Object Array Elements

If you have an array of Java objects, you can reference particular elements within that array with the `GetObjectArrayElement()` function. The function returns a single `jobject` that represents the Java object located at the specified array index. This `jobject` can be manipulated like any other Java object. Because `GetObjectArrayElement()` returns a reference to an object, any operations that you perform on the reference actually update the object stored within the array, so there is no need to copy the object back to the array. This makes is very easy to update fields in an object in an array, for example.

When you want to replace an object in an array, however, you don't want to have to alter every field within the object. In this case, you can use the `SetObjectArrayElement()` function to set a given element within an array to a specific object. This function is also useful for swapping objects in an array, such as during a sort operation, or for resetting all the elements to a default object. Unlike with the primitive types, JNI does not provide any functions that extract all of the elements of an object array, so you'll want to use a looping construct if you need to manipulate an entire array of objects.

One thing to remember about object arrays is that you can specify a default object for all of the elements when you create an object array with `NewObjectArray()`. In this case, however, each element is a reference to the same object. Thus, if you create an array and then start manipulating elements, you may encounter unexpected results. For example, if we implemented the `SymbolStone.load()` method using the following logic, we would end up with every element being equal to the last loaded symbol stone!

```
    /** The return array of objects... */
    jobjectArray rv = NULL;
        (char *)env->GetStringUTFChars( filename, NULL );

    /** Load the data from file into C structures... */
    ...

    /** Default initial element */
    jobject aSymbolStone = ...;

    /** Allocate an array of jobjects... */
    rv = env->NewObjectArray( numRecords, arg, aSymbolStone );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    for ( int i = 0 ; i < numRecords ; i++ ) {
        /** Get the stone to work with... */
        jobject stone = env->GetObjectArrayElement( rv, i );

        /** Update the fields within the class */
        ...
      }

    return rv;
  }
```

To resolve this problem, we should initialise the array to having a `NULL` initial element and, for each element in the array, create a new Java object. This removes the reference problem completely. For example:

```
    /** The return array of objects... */
    jobjectArray rv = NULL;

    /** Load the data from file into C structures... */

    /** Allocate an array of jobjects... */
    rv = env->NewObjectArray( numRecords, arg, NULL );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    for ( int i = 0 ; i < numRecords ; i++ ) {
        /** Allocate a new object for this element in the array... */
        jobject stone = ...;

        /** Update the fields within the class */
        ...

        /** Stuff the object into the array */
        env->SetObjectArrayElement( rv, i, stone );
      }

    return rv;
  }
```

Manipulating Array Elements with RNI

With the RNI, it is extremely easy to access array elements because the Microsoft JVM does not internally fragment arrays into non-contiguous sections of memory. While some JVM implementations internally optimize array data by splitting it into separate chunks that are located in different areas of memory, the Microsoft JVM does not do this. All arrays are stored in the same form, as contiguous data, just like arrays allocated within C or C++. As a result, arrays are far faster to access and manipulate than with the JNI.

As I mentioned earlier in the chapter, all arrays in RNI are represented by a set of special data types, `HArrayOf`*`Type`*. Each of these types behaves in an identical way, with the only difference being the internal data type that the array contains. Referencing the array elements of the array is extremely simple. An `HArrayOf`*`Type`* variable is a pointer to a `struct` that contains a single variable called `body`. This is true for arrays of any data type, including arrays of Java objects. The variable `body` points to a contiguous section of memory of the size required by the number of elements in the array. This array elements can be referenced either via a pointer or in the style of C arrays.

With this understanding, we can now look at a full implementation of the `Dimension3D.getDimensions()()` method:

```
/**
 * Returns a 3 element float array containing the width, height and depth
 */
RNIEXPORT HArrayOfFloat * RNICALL
Dimension3D_getDimensions( HDimension3D *arg ) {

    /** Extract the pointer */
    Dimension3D_t *dim = (Dimension3D_t *)arg->_pData;

    /** The return array of objects... */
    HArrayOfFloat *rv = NULL;

    /** Safety check */
    if ( dim == NULL || (int)dim == -1 ) {
        return NULL;
      }

    /** Allocate the array */
    rv = (HArrayOfFloat *)ArrayAlloc( T_FLOAT, 3 );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array\n" );
        return NULL;
      }

    rv->body[0] = dim->width;
    rv->body[1] = dim->height;
    rv->body[2] = dim->depth;

    return rv;
  }
```

All array types in RNI work in this manner. The only difference that you should expect to see when manipulating arrays of objects, arrays of arrays, or arrays of strings is that the techniques used to manipulate each element are different. However, each element is, of course, simply an object of that type and the usual rules apply there also.

Multi-dimensional Arrays

So far in this chapter, we've been talking about how to interact with and manipulate one-dimensional arrays within native code. But multi-dimensional arrays, or arrays of arrays, are fairly common, so now it's time to talk about them. For example, you might delare a 4x4 matrix as follows:

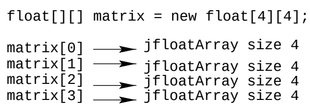
```
float[][] matrix = new float[4][4];
```

The techniques for creating and manipulating multi-dimensional arrays in native code rely on the basic one-dimensional array handling techniques, with a few conceptual tweaks that are applicable to both the JNI and RNI.

Multi-dimensional arrays with JNI

Creating a multi-dimensional array with the JNI is straighforward, since the JNI defines a `jarray` as a subtype of `jobject`. This allows you to declare the outer array as a `jobjectArray`, with each element being a `jarray` of a particular data type, as illustrated in Figure 7-1.

Figure 7-1. Multi-Dimensional JNI Arrays



For example, the `matrix` array defined earlier can be created within JNI with the following stub of code:

```
JNIEXPORT jobjectArray JNICALL
Java_matrix2_createArray( JNIEnv *env, jobject arg ) {

    int i = 0,
        j = 0;
    /** Create the “outer” array of objects, length 4... */
    jobjectArray outerArray =
        env->NewObjectArray( 4, env->FindClass( "java/lang/Object" ),
                             NULL );

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < env->GetArrayLength( outerArray ) ; i++ ) {
        /** Create a new array of floats... */
        jfloatArray floatArray = env->NewFloatArray( 4 );

        /** Extract the array elements for this subarray */
        jfloat *floatArrayElements =
            (jfloat *)env->GetFloatArrayElements( floatArray, NULL );

        /** Set them... */
        for ( j = 0 ; j < env->GetArrayLength( floatArray ) ; j++ ) {
            floatArrayElements[j] = ( i * 10 ) + j;
          }

        /** Tidy up... */
        env->ReleaseFloatArrayElements( floatArray, floatArrayElements, 0 );

        /** Set the new float array into the outer array */
        env->SetObjectArrayElement( outerArray, i, floatArray );
      }

    return outerArray;
  }
```

Since we are creating multi-dimensional array of a primitive data type, we can create the outer array of objects using the basic `java.lang.Object` class. This works because the actual object array elements are explicitly set to being an array anyway. Since `NewObjectArray()` requires a class to be given, it makes sense to use the superclass of all Java classes.

Creating a multi-dimensional array of actual objects, such as `SymbolStone` or `String` objects is even easier. Again, you create the outer array as an object array of type `java.lang.Object`, but each subarray is created as another `jobjectArray` that is manipulated in the normal way. For example, here’s some code that creates a 4x4 multi-dimensional arrays of `String` objects:

```
JNIEXPORT jobjectArray JNICALL
Java_matrix3_createArray( JNIEnv *env, jobject arg ) {
    int i = 0,
        j = 0;


    /** Create the “outer” array of objects, length 4... */
    jobjectArray outerArray =
        env->NewObjectArray( 4, env->FindClass( "java/lang/Object" ),
                             NULL );

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < env->GetArrayLength( outerArray ) ; i++ ) {
        /** Create a new array of floats... */
        jobjectArray stringArray =
            env->NewObjectArray( 4, env->FindClass( "java/lang/String" ),
                                 NULL );

        /** Set them... */
        for ( j = 0 ; j < env->GetArrayLength( stringArray ) ; j++ ) {
            /** Create a new string... */
            char s[1024];
            sObjectprintf( s, "String_%d%d", i, j );

            /** Create the Java String and set it in the array */
            jstring string = env->NewStringUTF( s );
            env->SetObjectArrayElement( stringArray, j, string );
          }

        /** Set the new float array into the outer array */
        env->SetObjectArrayElement( outerArray, i, stringArray );
      }

    return outerArray;
  }
```

Referencing the elements within an existing multi-dimensional array in native code is simply the reverse of creating the array. You iterate through the elements of the object array one by one, extracting the object and casting it to the appropriate array data type. Extraction and manipulation of those elements is now just a standard one-dimensional array issue. For example, the following native method takes a multi-dimensional array of floating point numbers:

```
JNIEXPORT void JNICALL
Java_matrix1_printArray( JNIEnv *env, jobject arg,
                         jobjectArray array ) {
```

```
    int i = 0,
        j = 0;

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < env->GetArrayLength( array ) ; i++ ) {
        /** Extract the array stored within this array... */
        jfloatArray floatArray =
            (jfloatArray)( env->GetObjectArrayElement( array, i ) );

        /** Extract the array elements for this subarray */
        jfloat *floatArrayElements =
            (jfloat *)env->GetFloatArrayElements( floatArray, NULL );

        /** Iterate through them and print... */
        for ( j = 0 ; j < env->GetArrayLength( floatArray ) ; j++ ) {
            fprintf( stderr, "[%d][%d]: %f\n", i, j, floatArrayElements[j] );
          }

        /** Tidy up... */
        env->ReleaseFloatArrayElements( floatArray, floatArrayElements, 0 );
      }

    return;
  }
```

Manipulating a multi-dimensional array of objects follows the same principle. The only difference is in the way in which you extract the "inner" array elements.

While we’ve only looked at two-dimensional arrays in this section, the principles we’ve used can be applied to three-dimensional arrays and beyond.

Multi-dimensional arrays with RNI

The RNI takes a similar approach, by allowing multi-dimensional arrays to be defined as elements of an `HArrayOfArray`. For example, a `float[4][4]` array can be composed of an `HArrayOfArray`, where each array element is an `HArrayOfFloat`. Here’s some code for creating a multi-dimensional array of floating point values:

```
RNIEXPORT HArrayOfArray * RNICALL
matrix2_createArray( Hmatrix2 *arg ) {

    int i = 0,
        j = 0;


    /** Create the “outer” array of objects, length 4... */
    HArrayOfArray *outerArray =
        (HArrayOfArray *)ClassArrayAlloc( T_CLASS, 4, "java/lang/Object" );

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < obj_length( outerArray ) ; i++ ) {
        /** Create a new array of floats... */
        HArrayOfFloat *floatArray =
            (HArrayOfFloat *)ArrayAlloc( T_FLOAT, 4 );

        /** Set them... */
        for ( j = 0 ; j < obj_length( floatArray ) ; j++ ) {
            floatArray->body[j] = (float)( i * 10 ) + j;
          }

        /** Set the new float array into the outer array */
        outerArray->body[i] = (HObject *)floatArray;
      }

    return outerArray;
  }
```

Creating a multi-dimensional array of Java objects with the RNI is just as easy:

```
RNIEXPORT HArrayOfArray * RNICALL
matrix3_createArray( Hmatrix3 *arg ) {

    int i = 0,
        j = 0;


    /** Create the “outer” array of objects, length 4... */
    HArrayOfArray *outerArray =
        (HArrayOfArray *)ClassArrayAlloc( T_CLASS, 4, "java/lang/Object" );

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < obj_length( outerArray ) ; i++ ) {
        /** Create a new array of floats... */
        HArrayOfObject *stringArray =
            (HArrayOfObject *)ClassArrayAl-
loc( T_CLASS, 4, "java/lang/String" );
```

```
        /** Set them... */
        for ( j = 0 ; j < obj_length( stringArray ) ; j++ ) {
            /** Create the string */
            char s[1024];
            sprintf( s, "String_%d%d", i, j );

            /** Set it within the array */
            stringArray->body[j] =
                (HObject *)makeJavaString( s, strlen( s ) );
          }

        /** Set the new float array into the outer array */
        outerArray->body[i] = (HObject *)stringArray;
      }

    return outerArray;
  }
```

As with the JNI, referencing the elements stored within a multi-dimensional is simply a case of reversing the operations used to create a new multi-dimensional array. The multi-dimensional array is treated as an array of object arrays, and for each of those elements, you simply cast it to an array of the appropriate type. The following code illustrates how to extract the values stored within a 4x4 matrix of floating-point numbers:

```
RNIEXPORT void RNICALL
matrix1_printArray( Hmatrix1 *arg, HArrayOfArray *array ) {

    int i = 0,
        j = 0;

    /** Iterate through each “outer” array element */
    for ( i = 0 ; i < obj_length( array ) ; i++ ) {
        /** Extract the array stored within this array... */
        HArrayOfFloat *floatArray =
            (HArrayOfFloat *)array->body[i];

        /** Iterate through them and print... */
        for ( j = 0 ; j < obj_length( floatArray ) ; j++ ) {
            fprintf( stderr, "[%d][%d]: %f\n", i, j, floatArray->body[j] );
          }
      }

    return;
    }
```

Referencing the elements within a multi-dimensional arrays of object is virtually identical, except that each subarray is of type `HArrayOfObject` and each element of that array is a Java object and should be treated as such.

Again, as with the JNI, the principles we've seen here with two-dimensional arrays in RNI can be applied to three-dimensional arrays and beyond.

Sizing Arrays

In all the array examples we've seen so far, we've assumed that we know the length of the array we are working with. Knowing the length is important, as it keeps you from running off the end of the array and throwing an `ArrayIndexOutOfBoundsException`. But if an array gets passed into a native method, you aren't likely to know its length in advance.

Fortunately, and not suprisingly, both the JNI and RNI provide a function that lets you determine the length of a Java array from within native code. With the JNI, this function is called `GetArrayLength()` and it works for both arrays of objects (and arrays) and arrays of primitive data types. This function returns the number of elements within the given array; it operates in exactly the same way as quering the value of the `length` variable implicitly associated with every Java array object. For example,

```
        jfloatArray aFloatArray = ...;

        for ( int j = 0 ; j < env->GetArrayLength( aFloatArray ); j++ ) {
            /* Do something to each element */
            ...
          }
    }
```

The corresponding RNI function is called `obj_length()`. This function correctly counts the number of elements within any type of array, including arrays of primitive data types and arrays of objects (and arrays). For example:

```
        HArrayOfFloat *aFloatArray = ...;

        for ( int j = 0 ; j < obj_length( aFloatArray ); j++ ) {
            /* Do something to each element */
            ...
          }
```

```
}
```

Notes

1. `ClassArrayAlloc()` can also be used to allocate an array of a primitive data type, by passing the same parameters as with `ArrayAlloc()`. In this case, the final parameter should be `NULL`.

Chapter 8. Classes and Objects with the JNI

Because Java is an object-oriented language, classes and objects are central in every Java program. Objects, which are instantiations of classes, form the basic building blocks that your programs use to manipulate data. Thus, it goes without saying that you need to be able to manipulate classes and objects in your native code. Since the JNI and the RNI have very different behaviors with regards to working with classes and objects, we're going to focus exclusively on the JNI in this chapter. Chapter 9 discusses class and object manipulation with the RNI.

In Chapter 4, I mentioned that the JNI requires you to specify the class of a given Java object in order to locate specific pieces of information within that class or perform operations on it. To access a field or call a method within an object or class, you need a value of type `jclass` that contains the class information for the class in question. Thus, being able to locate `jclass` values for relevant classes is a prerequisite for accessing fields and calling methods of Java classes and objects within native code. The JNI provides several ways for you to get `jclass` values.

There are a number of object- and class-based operations that you perform on a routine basis within Java that you might want to perform in native code. For example, you might need to check that an object of one class is castable to another class. Or you might want to find out whether two classes have a common superclass. These operations are possible within native code using the JNI.

This chapter also discusses how to perform common object actions from within native code: retrieving and setting values of fields defined within a Java class, invoking methods defined within a Java class, and creating new Java objects.

Locating Classes

The JNI defines three functions that let you locate `jclass` values that represent Java classes. One function takes the name of a class file and attempts to scan any classes on your hard disk for the class information. Another function gives you the class information for an existing Java object. The final method allows you to construct a new Java class from an arbitrary stream of data, such as raw data coming in over a network connection.

`FindClass()` is the most commonly used function. It scans the directories and ZIP files defined in the `CLASSPATH` environment variable for a given class, and, if successful, returns the `jclass` value for that class. `FindClass()` takes one argument, the fully qualified class name of the class for which you are searching. For example, the `String` class has the fully qualified name of `java/lang/String`, where `java/lang` is the package that the class belongs to. While the period (`.`) is used as a separator in class names in Java code, JNI functions expect a slash (`/`) character instead.

As of Java 2 SDK Version 1.2, the functionality of `FindClass()` has been extended in an important way. Prior to this version, `FindClass()` could only locate local classes. Now the function can also locate classes that have been loaded from a remote location with a customized `ClassLoader`.

If successful, `FindClass()` returns a `jclass` value for the desired class. Upon failure, however, the function simply returns `NULL`. This can occur if the class cannot be located within the `CLASSPATH` or if the class can be located but is corrupted or fails the byte code verification stage. If the function fails, it throws an exception that specifies the exact cause of the problem.

Now let's look at an example of using `FindClass()`. Each symbol stone has the possibility of having some names that it is alternatively known as. We can retrieve these names, if they exist, using the `SymbolStone.getAlternativeNames()` method which returns an array of `String`s. Therefore, before we create any of the `String`s to return we need to locate the `jclass` value for the `java/lang/String` class.

```
    jclass stringClass = env->FindClass( "java/lang/String" );
    if ( stringClass == NULL ) {
        fprintf( stderr, "Failed to locate class information!\n" );
        return NULL;
      }

    fprintf( stderr, "Located class information!\n" );

    ...
```

`FindClass()` is a useful function when you know in advance exactly which classes you want to manipulate within native code. If you don't know the names of the classes of interest in advance, however, or you have native code that performs operations on objects without really caring which class they belong to, `FindClass()` isn't the right function.

If you have an object of the class you need to manipulate, it is much easier to use the JNI function `GetObjectClass()` instead. This function simply returns a `jclass` value that represents the class from which the object was instantiated. Say you have a Java object that is passed into native code as an argument to a method. `GetObjectClass()` makes it easy to get the `jclass` value, so that you can access fields or call methods within that object. Here's a short example of using `GetObjectClass()` in just this way:

```
Java_Class_method( JNIEnv *env, jobject arg, jobject targetObject ) {

    /* Get the class information for targetObject */
    jclass targetObjectClassblock = env->GetObjectClass( targetObject );

    /* Now you can access fields and call methods of targetObject */
    ...
  }
```

`DefineClass()` is the final function that produces a `jclass` value and it is far more tricky to use than the functions we've already discussed. This function lies at the heart of Java's dynamic extensibility.

`DefineClass()` reads a stream of data from an external source, such as a file or network connection, and creates a Java class from it. The data stream must contain valid Java byte codes (i.e., the contents of a class file). The `DefineClass()` function itself is quite simple: you just pass a a buffer that contains valid Java byte codes and it returns a `jclass` value for that class. What's complex is defining the infrastructure of transferring the data into the buffer in the first place.

`DefineClass()` relies on the `ClassLoader` mechanism supplied in standard Java to load foreign Java classes into the JVM. For example, a VRML scene might contain a `Script` node, which is a piece of logic written in Java that drives part of an animation of the scene. A VRML browser written in Java might use a `ClassLoader` to download the class file data from the given location and execute the resulting Java class to drive the scene's animation.

Similarly, a `ClassLoader` can be used to dynamically extend the abilities of Java. If you loaded a dumb TV-set-like device with a JVM and a `ClassLoader`, you could conceivably download Java code over the standard television channels and run it on your TV set (or your toaster or any other device that could use Java). This is why Java's dynamic extensibility makes it so powerful.

Creating a new `ClassLoader` is a privileged activity, which means that applets cannot do it. To determine whether your application can create a new `ClassLoader`, your program should invoke the following lines of code:

```
try {
    /* Check to see if we can create a new ClassLoader */
    System.getSecurityManager().checkCreateClassLoader();
  } catch ( SecurityException e ) {
    System.err.println( "ClassLoader cannot be created! " + e.toString() );

    /** Abort the DefineClass() procedure! */
  } catch ( NullPointerException e2 ) {
    System.err.println( "No security manager present!" );

    /** This isn't fatal, so keep going... */
  }
```

The first step in using native code and `DefineClass()` to create a new class is to define a custom `ClassLoader` in Java. The following class, called `testClassLoader`, simply subclasses `ClassLoader` and overrides the `loadClass()` method that returns the class information of the new class. In this case, `loadClass()` calls a native method that returns the valid class data:

```
public class testClassLoader extends ClassLoader {
    /** static initializer block to load the native library */
    static {
        try {
            System.loadLibrary( "defineclass1" );
          } catch ( UnsatisfiedLinkError e ) {
            System.err.println( "Cannot load defineclass1 native library: " +
                                e.toString() );
          }
      }

    /** Native method that defines a class */
    private native Class defineClass( String classname );

    /** The loadClass method */
    public synchronized Class loadClass( String name, boolean resolve ) {
        Class c = defineClass( name );
        return c;
      }
  }
```

The next step is using the custom `ClassLoader` within your Java code. For example:

```
    /**
     * Check the SecurityManager that we can create a new
     * ClassLoader.
     */
    try {
        System.getSecurityManager().checkCreateClassLoader();
        System.err.println( "We can create a new ClassLoader!" );
      } catch ( SecurityException e ) {
        System.err.println( "We cannot run this example since we cannot cre-
ate a new ClassLoader!" );
        System.exit( 1 );
      } catch ( NullPointerException e ) {
        System.err.println( "No security manager!" );
      }

    testClassLoader classLoader = new testClassLoader();

    /** Invokes the native method */
    Class c = class-
Loader.loadClass( "../../../classes/testObject.class", true );

    /** Print out the results to see if the class has loaded OK... */
    if ( c != null ) {
        System.err.println( c.toString() );
      }

    ...
```

Finally, here's the implementation of the native method that actually reads the class data from a file, defines the new class, and returns the `jclass` value:

```
/**
JNIEXPORT jclass JNICALL
Java_testClassLoader_defineClass( JNIEnv *env, jobject arg,
                                  jstring className ) {

    int i;

    /** Classblock info for the class we're creating from bytecode */
    jclass testClass;

    /** Buffer to 'stat()' the file into */
    struct stat classStat;

    /** The buffer to read the bytecode into */
    const jbyte *bytecodeBuffer;

    /** Read in the raw data for the test class from disk */
    FILE *classFile;

    fprintf( stderr, "here\n" );

    /** Get the filename containing the bytecode... */
    const char *classNameChars = env->GetStringUTFChars( className, NULL );

    classFile = fopen( classNameChars, "r" );
    if ( classFile == NULL ) {
        fprintf( stderr, "Failed to open %s for reading!\n", class-
NameChars );

        /** Deallocate the class name String */
        env->ReleaseStringUTFChars( className, classNameChars );

        return NULL;
      }
    /** Get the length of the bytecode */
    if ( stat( classNameChars, &classStat ) != 0 ) {
        fprintf( stderr, "Error occurred in stat()’ing %s!\n", class-
NameChars );

        /** Deallocate the class name String */
        env->ReleaseStringUTFChars( className, classNameChars );

        return NULL;
      }

    /** Create the buffer to read the bytecode into */
    fprintf( stderr, "Allocating buffer of %d bytes\n",
                     ( classStat.st_size * sizeof( jbyte ) ) );
    bytecodeBuffer =
        (const jbyte *)malloc( classStat.st_size * sizeof( jbyte ) );
    memset( bytecodeBuffer, 0, classStat.st_size * sizeof( jbyte ) );

    /** Read the bytecode from the file */
    if ( fread( (jbyte *)bytecodeBuffer, sizeof( jbyte ),
                classStat.st_size, classFile ) != classStat.st_size ) {
        fprintf( stderr, "Error occurred in reading %s!\n", classNameChars );
        return NULL;
      }

    /** Check the magic header of the byte-
code. It should equals 0xCAFEBABE */
    fprintf( stderr, "Checking magic header: 0x" );
    for ( i = 0 ; i < 4 ; i++ ) {
        fprintf( stderr, "%X", bytecodeBuffer[i] & 0xff );
      }
    fprintf( stderr, "\n" );

    /**
     * Now that we have read the bytecode for the test class
     * into an appropriate buffer, try a use ’DefineClass()’ to extract
     * the classblock of it.
     */
    testClass =
        env->DefineClass( (const char*)"testObject",
                          NULL, bytecodeBuffer,
                          classStat.st_size );
    if ( testClass != NULL ) {
        fprintf( stderr, "Located classblock from defined class!\n" );
      } else {
        fprintf( stderr, "Failed to locate classblock from de-
fined class!\n" );
      }

    /** Deallocate the class name String */
    env->ReleaseStringUTFChars( className, classNameChars );

    /** Deallocate the bytecode buffer */
    free( (jbyte *)bytecodeBuffer );

    return testClass;
  }
```

The second argument to `DefineClass()` allows you to specify the `ClassLoader` with which the class should be loaded. If your native code is being called from a custom `ClassLoader`, it is better to specify `NULL` to avoid circularity. However, when a standard `ClassLoader` can be used, you may specify an instance of it with this argument.

Being able to instantiate Java classes from native code adds additional flexibility to Java's class loading mechanisms. The Java `ClassLoader` is restricted in that it can only load Java class files from sources that can be established using other Java classes, which limits the sources to files and network connections. With `DefineClass()`, however, you can write some extremely powerful and self-expanding software that can seamlessly use underlying hardware devices, such as infrared beams and television signals, as the source for Java class information.

Testing Class and Object Characteristics

Within Java, there are certain core operations you can perform on classes and objects. For example, you can use the `instanceof` keyword to determine whether a given object is an instance of a certain class and the = operator to see if two object references point at the same object. To make life easier for you, the JNI defines several functions that let you perform these same operatons on classes and objects in native code.

The JNI functions for operating on classes are `GetSuperclass()`, which returns a `jclass` value that represents the superclass of the given class, and `IsAssignableFrom()`, which determines whether an object of one class can be safely cast to another class. For working with objects, there is `IsInstanceOf()`, which tests to see whether a given object is an instance of the given class, and `IsSameObject()`, which tests whether two object references refer to the same Java object.

Accessing Fields

Accessing the values of fields within Java objects is an extremely common operation in Java programming. Given that native methods are simply an extension of standard Java code, native code needs direct access to fields as well. It is also quite common for a Java method to update the values of a number of fields within an object. If a native method calculates new values for these fields, it should be able to update them directly too.

Without the ability to access and update fields directly, your native code would start to resemble spaghetti. You would need to pass many more arguments to your native methods, in place of reading the fields from native code. You would also need to invoke multiple native methods, each returning a single value, in order to update multiple fields to new values. This would make native methods rather cumbersome, to say the least.

Fortunately, the JNI defines functions that let you both access and update the values of fields within Java classes (class, or `static`, fields) and fields within objects instantiated from those classes (instance fields).

Field Identifiers

Before you can access the value stored within a field of a Java class or object, you have to get a *field identifier* that uniquely identifies the field. This identifier is based on the class in which the field is declared, as well as the name and type signature of the field. This might seem like overkill, but consider the implications if we don't specify all this information. By specifying the type signature, we can ensure that we get an identifier for the exact field we want, not a field with the same name but different type that is defined in a superclass, for example.

Thus, the first step in accessing a field is getting its unique identifier, which is a `jfieldID` value. Of course, to get a `jfield` value, you first have to obtain a `jclass` value for the class in which the field is declared, as we discussed in the previous section. You also need to know the name of the field that you want to access and its type signature. For more information about type signatures, see Chapter 4.

The JNI defines two functions for getting a unique identifier for a field: `GetFieldID()` and `GetStaticFieldID()`. `GetFieldID()` locates an instance field, while `GetStaticFieldID()` locates a class (`static`) field. Thus, you have to know which kind of field you are dealing with, in order to call the appropriate function. Beyond that, both methods take exactly the same arguments and both return a `jfieldID` value that represents a unique field identifier. Just be sure that you call the right function, depending on whether you are dealing with an instance or class field, otherwise you may get unpredictable results.

For example, in most of the symbol stone native methods, we tend to retrieve an integer value from the `SymbolStone` object stored in the `_pData` field as follows:

```
/* Returns the name of the stone  */
JNIEXPORT jstring JNICALL
Java_SymbolStone_getName( JNIEnv *env, jobject arg ) {

    /** Extract the pointer... */
    jfieldID pDataField =
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" );

    ...
```

Another way that we can locate a field identifier from a class is to use the Reflection API. This allows us to "look into" the definition of a class and extract information about field, methods and other bits of information. This sort of thing can be useful if you don't know in advance the names of fields that you might want to extract information from, for example, if you were implementing a debugger or class file inspector.

For example, we can use the `Class.getFields()` method to return an array of `java.lang.reflect.Field` objects, one per field in the class. With the JNI, we can go one step further and convert the `Field` object to a `jfieldID` for use within our native methods. For example:

```
    ...

    /** Invoke the Class.getFields() method...Returns an array of Field */
    jobjectArray fieldsArray =
        (jobjectArray)env->CallObjectMethod( env-
>GetObjectClass( arg ), getFieldsMethodID );
    if ( fieldsArray == NULL ) {
        fprintf( stderr, "Failed to invoke getFields()\n" );
        return;
      }

    /** Convert the fields from java/lang/reflect/Field to jfieldID... */
    for ( int i = 0 ; i < env->GetArrayLength( fieldsArray ) ; i++ ) {
        jfieldID fieldID =
            env->FromReflectedField( env->GetObjectArrayElement( fieldsArray,
                                                                 i ) );

        /** Display the value of the field... */
        ...
      }

    ...
```

Similarly, if we wished to, we can convert the `jfieldID` value to a `java/lang/reflect/Field` object by using the corollary `ToReflectedField()` JNI method.

And, now that we have located the unique identifier pointing to the field of interest, we can look into how we manipulate the value stored within it.

Manipulating Field Data

There are two operations you can perform upon fields within a class: getting the field data and setting the field data. In other words, you can retrieve the current value of a field and set a field to a new value. These two operations allow your native code to read data that has been created or calculated in Java and pass data directly back into Java objects.

One of the common questions asked by neophyte native method programmers is “How can I store a C struct or C++ class within a Java object?” If you are using native code to tie some legacy code to a Java front end, it can be quite useful to store pointers to native data structures within your Java classes, as we’ve seen with our example. This may seem to be a conundrum though, since C and C++ programs use pointers to track this type of information, but Java doesn’t have pointers!

What you need to remember is that that a pointer is basically just a number that represents a memory address. Thus, it’s easy to store a pointer within a Java object in a variable declared as an `int`.[1] Within your native code, if you need to store a C `struct` and keep track of it in your Java code, you can simply use `malloc()` to allocate the memory for the `struct` and then set the value of the Java `int` field to the memory address at which the `struct` is located. Later, when you need to recover your original data structure, you only need to extract the `int` value from the Java object and cast it back to a pointer to the `struct`. Not suprisingly, this is exactly the technique that the symbol stone example code uses!

Retrieving Field Values

Once you have a field ID for a Java instance field, you can get the value of the field using one of the `Get`*`Type`*`Field()` functions. The JNI provides a function for every primitive data type, plus a single function, `GetObjectField()`, for object-type fields. Each of these functions takes an object reference (`jobject`) and a field ID and returns the value of the field as the appropriate type. You need to be sure to call the `Get`*`Type`*`Field()` function that matches the type of the instance field you are working with.

If you are dealing with a class (`static`) field, use the `GetStatic`*`Type`*`Field()` functions instead. Again, there is a function for every primitive data type, plus a single function, `GetStaticObjectField()`, for object-type data. The only difference between these functions and the ones for instance fields is that each `GetStatic`*`Type`*`Field()` function takes a `jclass` value as its first argument, instead of a `jobject` value. The second argument is still a field ID, although it should be a field ID that refers to a `static` field, of course. As with instance fields, each function returns the value of the class field as the appropriate type.

As we've already discussed, almost every native method in the symbol stone example needs to access the `_pData` field in a Java class. This field contains a pointer to the appropriate legacy data structure. Here's part of the code for the JNI implementation of the `SymbolStone.getName()` method that shows how to extract the field value and cast it to a `SymbolStone_t` value:

```
/** Returns the name of the stone */
JNIEXPORT jstring JNICALL
Java_SymbolStone_getName( JNIEnv *env, jobject arg ) {
    /** Extract the pointer */
    jfieldID pDataFieldID = \
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" ); \
    SymbolStone_t *stone = \
        (SymbolStone_t *)env->GetIntField( arg, pDataFieldID );

    /** Safety check */
    if ( stone == NULL || (int)stone == -1 ) {
        return NULL;
      }

    /** Create a new Java String from the name */
    return env->NewStringUTF( stone->name );
  }
```

If you are accessing a field that contains an object reference, you need to use either `GetObjectField()` or `GetStaticObjectField()`. The return type of each function is `jobject`, so you need to cast the returned value to the appropriate object type before you use it in your native code.

These two functions can also be used to retrieve a Java array from a Java class or object, which is important because the JNI does not provide separate functions for accessing array fields. If you recall our discussion of JNI data types from Chapter 4, the data type that represents Java arrays, `jarray`, is a "child" of the `jobject` data type. Thus it is perfectly legitimate to use the `GetObjectField()` and `GetStaticObjectField()` functions to get the value of an array field from Java. All you have to do is cast the returned value to the appropriate array data type and then you can use the array as described in Chapter 7.

Setting Field Values

The JNI functions for setting field values follow the same format as the functions for retrieving field values: there are separate functions for setting both instance and class fields for each of the primitive data types, as well as the generic Java object type, `jobject`. As you might expect, these functions take the form `Set`*`Type`*`Field()` and `SetStatic`*`Type`*`Field()`.

These methods are syntactically identical except that in the case of the static variable methods the class in which you wish to set the field's value is specified instead of the Java object.

To set the value of an instance field within a Java object, you simply need to invoke the appropriate function, passing the object reference (`jobject`, the field ID, and the new field value. For a class (`static`) field, the only difference is that you pass a `jclass` value instead of a `jobject`. For example, to set the value of an instance `int` field from native code, you use the `SetIntField()` function and specify the new value as a `jint`. Passing a value of a different data type than what is expected by the method (e.g., passing a `float` to `SetIntField()`) can have unpredictable results. The most likely effect is that the value will suffer the usual consequences of casting between the two data types, but in extreme cases the JVM itself may panic and abort.

Here's the JNI implementation of the `SymbolStone.load()` native method that shows how the `_pData` field is set to a pointer to the corresponding legacy `SymbolStone_t` structure:

```
    ...

    for ( int i = 0 ; i < numRecords ; i++ ) {
        /** Create a new SymbolStone object */
        jobject stone = ...;
        if ( stone == NULL ) {
            fprintf( stderr, "failed to construct new instance of Symbol-
Stone class\n" );
            return NULL;
          }

        /** Set the pData pointer... */
        jfieldID pDataFieldID =
            env->GetFieldID( arg, "_pData", "I" );
        if ( pDataFieldID == NULL ) {
            fprintf( stderr, "failed to locate fieldID for pData\n" );
            return NULL;
          }
        env->SetIntField( stone, pDataFieldID, (jint)sptr );

        ...
      }

    ...
```

Just as with the object field retrival functions, the `SetObjectField()` and `SetStaticObjectField()` functions can be used to set the value of an object- or array-type Java field. This is possible because the `jarray` JNI data type is castable to `jobject`, the generic Java object data type.

Invoking Methods

Even more so than accessing fields, you are going to want to invoke methods on Java objects and classes from your native code. That's because almost all functionality in a Java program happens as a result of method calls. In fact, you can typically think of a Java program as a series of "cascading method invocations", where a method calls another method, which in turn calls another method, and so on. For example, our symbol stone example creates instances of `SymbolStone` objects, which in turn contain `Dimension3D` objects. If you want to print out the dimensions of each stone, it can invoke the `toString()` method in the `SymbolStone` object which might call the `toString()` method of the relevant `Dimension3D`.

Without the ability to invoke Java methods from native code, there really wouldn't be any point to linking Java code with native code. Being able to call Java method allows you to take advantage of Java functionality in your native code, instead of having to reimplement the functionality in native code. The whole point of combining Java code with native code is to support the use of legacy code, while taking advantage of the strengths of Java.

Given the importance of being able to invoke Java methods, it shouldn't suprise you to find out that the JNI defines functions that let you invoke methods on an object (instance methods) and methods on a class (class, or `static` methods).

Method Identifiers

Just as with accessing fields, the first step in invoking a Java method from native code is getting a *method identifier* that uniquely identifies the method. This identifier is based on the class in which the method is declared, as well as the name and type signature of the method. These three parameters uniquely identify the method you want to invoke. The type signature is particularly important in identifying methods, given that Java supports overloaded methods, whereby multiple methods can have the same name but different parameters and return types.

For example, a 3D graphics library might define several methods to specify a point in two- or three-dimensional space. Here are some overloaded method declarations for specifying a point:

```
void vertex( int x, int y )
void vertex( float x, float y )
void vertex( int x, int y, int z )
void vertex( float x, float y, float z )
```

These four methods are all defined in the same class and they have the same name, so you can see why we need to specify a type signature to get the method we want.

The JNI provides two functions for retrieving method identifiers: `GetMethodID()` for instance methods and `GetStaticMethodID()` for class (`static`) methods. Each function takes `jclass` value for the class in which the method is declared, the name of the method, and its type signature, and returns the unique identifier of type `jmethodID` for the desired Java method, provided that the method exists. You can get the appropriate `jclass` value using `FindClass()` or `GetObjectClass()`, as we discussed earlier. For more information about type signatures, see Chapter 4. Finally,be sure that you call the right function, depending on whether you are dealing with an instance or class method, otherwise you may get unpredictable results.

For example, here is a code fragment that shows how to locate the method ID for each of the `vertex()` methods we declared previously:

```
/* The vertex() methods are defined within a class called Point */
jclass PointClassblock = env->FindClass( "Point" );

/* Locate the method IDs for each method variant */
jmethodID vertex2intsMethodID =
    env->GetMethodID( PointClassblock, "vertex", "(II)V" );

jmethodID vertex2floatsMethodID =
    env->GetMethodID( PointClassblock, "vertex", "(FF)V" );

jmethodID vertex3intsMethodID =
    env->GetMethodID( PointClassblock, "vertex", "(III)V" );

jmethodID vertex3floatsMethodID =
    env->GetMethodID( PointClassblock, "vertex", "(FFF)V" );
```

Note the use of character strings to specify the method name and type signatures. That’s all there is to retrieving method IDs.

The `GetMethodID()` and `GetStaticMethodID()` functions return a `jmethodID` value if the function call succeeds. Of course, these functions can fail for a number of reasons, including misspelled method names or invalid type signatures. In cases of failure, the function returns a `NULL` value and throws an exception that specifies the reason for failure.

We can also use the Reflection API to locate method identifiers in a way similar to that discussed earler regarding fields. For example, we can invoke `Class.getMethods()` which will return an array of `java/lang/reflect/Method` objects. We could convert those objects to `jmethodID` values using the JNI `FromReflectedMethod()` method. Or, convert them vice versa using `ToReflectedMethod()`.

Furthermore, as constructors are also represented by `jmethodID` values, `To/FromReflectedMethod()` will also convert to and from `java/lang/reflect/Constructor` objects.

Invoking Instance Methods

When you invoke an instance method on an object, you are typically calling the method that is defined by the class of that object. However, in some cases, you may want to invoke the method defined by the superclass of the object, performing the native code equivalent of using the `super` keyword in Java. In this section, we're just going to look at invoking instance methods defined by the object in question; we'll get to details on invoking superclass methods in the next section.

The JNI defines a whole slew of different functions for invoking instance methods. These functions come in three basic groups:

`Call`*`TypeMethod()`*

`Call`*`TypeMethodA()`*

`Call`*`TypeMethodV()`*

In each group of functions, *`Type`* refers to the return type of the method being called. There are separate functions for each of the primitive data types, for `jobject`, and for a return type of `void`.

The difference between each group of functions is in the way in which the arguments to be passed to the Java method are packaged up. The standard way of invoking a Java method from native code is with the `Call`*`Type`*`Method()` functions, which simply require that you pass arguments that match the parameters required by the Java method. For example, if we want to call the `vertex()` method that takes two integer parameters, we might use `CallVoidMethod()` as follows:

```
env->CallVoidMethod( aPoint, vertex2intsMethodID, 333, 444 );
```

In this case, `333` and `444` are the arguments to be passed to the Java method. This function call is equivalent to the following Java code:

```
vertex( 333, 444 );
```

The `Call`*`Type`*`MethodA()` functions take a slightly different approach to passing arguments to the Java method. With these functions, you pass in an array of `jvalue` values, where each element of the array contains an argument value. Thus, in the case of the `vertex(int x, int y)` method, we need to create and populate an array of two `jvalue` values.

One advantage of the `Call`*`Type`*`MethodA()` functions is that they provide optimized memory access to the arguments being passed in. With the `vertex()` method, for example, we are passing two separate `jint` values that could be located in totally different areas of memory, which means that it could take longer to access the values. If we specify the arguments as an array of `jvalue` values, however, the two integers are guaranteed to be contiguous in memory. When a method has many arguments and is called frequently, the processing time saved by using `Call`*`Type`*`MethodA()` can add up.

Another advantage with these functions is that you don't have to worry about explicitly casting arguments to the appropriate types before invoking the method. Setting a value within a `jvalue` ensures that it is set to the correct type prior to method invocation.

Here's an example of using `CallVoidMethodA()` to invoke the `vertex()` method that takes three `float` arguments:

```
/* Allocate an array of 3 jvalues to store the arguments in */
jvalue *argArray = (jvalue *)malloc( 3 * sizeof( jvalue ) );

/* Store the three argument values in the array */
argArray[0].f = 111.11;
argArray[1].f = 222.22;
argArray[2].f = 333.33;

/* Invoke the method passing the argument array */
env->CallVoidMethodA( aPoint, vertex3floatsMethodID, argArray );

/* Deallocate the array */
free( argArray );
```

Before you call `Call`*`Type`*`MethodA()`, you must create the array of `jvalue` values yourself—the JNI does not do this for you. And when you are done, you must also remember to deallocate the array or a memory leak will occur.

Recall that `jvalue` is a `union`, which means that you can set it to any primitive value or a `jobject` value. Here we've used the `f` segment, which corresponds to a `jfloat` value. If we wanted to set an integer value instead, we would use the `i` segment as follows:

```
argArray[0].i = 111;
```

The final group of method invocation functions, `Call`*`Type`*`MethodV()`, uses the standard ANSI C varargs interface to batch up the arguments. The varargs interface allows you to define a list of arguments dynamically; the argument list is encoded and decoded internally by the varargs interface. The varargs functions make sense when you require flexibility, such as when the number of arguments is not known at compile-time. Another benefit is that allocating and deallocating varargs lists is more friendly than manual allocation and deallocation of memory for arrays of `jvalue` values.

However, the downside to using the varargs interface is that it may not be portable between operating systems and even compilers. Each compiler tends to implement its own version of varargs, and, even though it is part of ANSI C, the actual implementation of varargs can vary greatly in quality across platform as well. Caveat emptor.

Invoking Methods of a Superclass

As I mentioned earlier, there are some situations where you want to be able to call the method defined in the superclass of an object, rather than the one defined in the class of the object itself. In Java, this can be done with the `super` keyword:

```
super.toString();
```

This code executes the `toString()` method defined in the superclass, instead of the method defined in the actual class of the object.

The ability to invoke the superclass' method comes into play when a Java class defines a method that overrides the method defined in the superclass. In this case, the overriding method may need to be able to call the method defined in the superclass, to take advantage of the functionality it provides. Suppose, for example, that our `vertex()` method overrides the method in its superclass. If we want to invoke the superclass method, we can do so as follows:

```
public void vertex( int x, int y ) {
    /* Do some stuff */
    ...

    /* Call the superclass method */
    super.vertex( x, y );
  }
```

Note that the `super` keyword can only be used this way within the overriding class, as we've shown here.

Within the JNI, the corresponding functionality is provided by three groups of functions:

`CallNonvirtual`*`TypeMethod()`*

`CallNonvirtual`*`TypeMethodA()`*

`CallNonvirtual`*`TypeMethodV()`*

Each group of function handles arguments in a particular way (i.e., a list of arguments, an array of `jvalue` values, and as varargs), just like the methods for invoking instance methods that we discussed in the previous section. And in each group, there is a separate function for each of the primitive data types, for `jobject`, and for a return type of `void`.

The nonvirtual invocation functions are used almost identically to their virtual counterparts, except that they all take an additional argument of type `jclass`.[2] The `jclass` value should represent the superclass in which the method you want to invoke is defined, while the `jobject` value specifies the actual object on which the method is to be invoked. In addition, the method ID you pass to a nonvirtual invocation function should specify the method *within the superclass*. One final warning is that the `jclass` value you specify as the superclass of the object really should be the superclass, or the JVM is likely to do something extremely unpredictable.

Invoking Class Methods

The JNI defines three more groups of functions for invoking class (`static`) methods:

`CallStatic`*`TypeMethod()`*

`CallStatic`*`TypeMethodA()`*

`CallStatic`*`TypeMethodV()`*

Each group handles method arguments differently, as we've already seen. And just as with the instance method invocation functions, each group has a separate function for each primitive type, for `jobject`, and for a return type of `void`.

The only difference between the functions that invoke instance methods and the ones that invoke class methods is that the later functions all take a `jclass` argument instead of a `jobject`. This argument specifies the class in which to invoke the `static` method.

Creating Objects

Objects are central to any Java program, so it is imperative that you be able to create Java objects from within native code. As you would expect, the JNI provides functions that allow you to do just that. One situation where it is useful to be able to create Java objects within native code is when you are manipulating arrays of objects and need to set array elements to new objects. Another example of object creation occurs when your legacy code creates a new object internally that needs to be reflected in Java. The JNI supports two basic techniques for object creation: invoking an object constructor or allocating a new object without invoking a constructor.

Invoking Object Constructors

The simplest way to create a new Java object from within native code is to execute one of the constructors defined by the object's class. This is exactly what you do to create a new object within Java itself. The JNI defines three functions for executing constructors that differ only in the way in which arguments to the constructor are specified:

`NewObject()`

`NewObjectA()`

`NewObjectV()`

These three functions mirror the JNI method invocation functions in their handling of arguments. The standard way of executing a constructor is with `NewObject()`, which simply requires that you pass arguments that match the parameters required by the Java constructor. `NewObjectA()` bundles the arguments into an array of `jvalue` values that are passed to the constructor in one contiguous block. `NewObjectV()` uses the standard ANSI C varargs interface to specify a single list of argument values for the constructor.

Each of these functions takes a `jclass` value and a method identifier specified as a `jmethodID` value, in addition to any arguments that are passed along to the constructor. The `jclass` specifies the class that you wish to create a new object of. You can get the appropriate `jclass` value using `FindClass()` or `GetObjectClass()`, as we discussed earlier.

The `jmethodID` argument may seem a bit puzzling, until you realize that constructors are just methods, albeit methods with a special purpose. So, before you can invoke a constructor, you have to call `GetMethodID` to get a `jmethodID` for the constructor. The only tricky part is specifying the "method name" of the constructor, since in Java constructors have the same name as the class. With the JNI, you use a method name of <`init`> for a constructor. The type signature is specified just as for any other method, except that the return type is always `void`.[3] Note that a class can have multiple constructors that take different arguments, just like methods can be overloaded, so it is important to specify the correct type signature.

`NewObject()`, `NewObjectA()`, and `NewObjectV()` each returns a `jobject` value that represents the new Java object, if the function executes successfully. If the function call fails for some reason, it returns `NULL` and throws an exception that specifies the reason for failure.

Now that we've discussed how to create objects, we can revisit the `SymbolStone.load()` native method we have discussed in passing throughout earlier chapters. This static method returns an array of `SymbolStone` objects created from an external data file and is implemented as follows:

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
JNIEXPORT jobjectArray JNICALL
Java_SymbolStone_load( JNIEnv *env, jclass arg, jstring filename ) {

    /** The return array of objects... */
    jobjectArray rv = NULL;

    /** Extract the filename */
    char *filenameChars =
        (char *)env->GetStringUTFChars( filename, NULL );
    /** Load the data from file into C structures... */
    int numRecords = 0;
    SymbolStone_t *stones = load( filenameChars, &numRecords );
    SymbolStone_t *sptr = stones;

    /** Release the filename */
    env->ReleaseStringUTFChars( filename, filenameChars );

    /** Safety check... */
    if ( stones == NULL || numRecords == 0 ) {
        fprintf( stderr, "stones array == NULL or %d records == 0\n", num-
Records );
        return NULL;
      }

    /** Otherwise, for each stone, allocate a new Java object and populate */
    jmethodID ctorMethodID =
        env->GetMethodID( arg, "<init>", "()V" );
    if ( ctorMethodID == NULL ) {
        fprintf( stderr, "Cannot locate constructor for Symbol-
Stone class!\n" );
        return NULL;
      }

    /** Allocate an array of jobjects... */
    rv = env->NewObjectArray( numRecords, arg, NULL );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    for ( int i = 0 ; i < numRecords ; i++ ) {
        jobject stone = env->NewObject( arg, ctorMethodID );
        if ( stone == NULL ) {
            fprintf( stderr, "failed to construct new instance of Symbol-
Stone class\n" );
            return NULL;
          }

        /** Set the pData pointer... */
        jfieldID pDataFieldID =
            env->GetFieldID( arg, "_pDataFieldID", "I" );
        if ( pDataFieldID == NULL ) {
            fprintf( stderr, "failed to locate fieldID for pData\n" );
            return NULL;
          }
        env->SetIntField( stone, pDataFieldID, (jint)sptr );

        /** Stuff the object into the array */
        env->SetObjectArrayElement( rv, i, stone );

        /** Increment the pointer... */
        sptr++;
      }

    return rv;
  }
```

Creating Objects without Constructors

Another way to create a Java object from within native code is to create the object *without* invoking a constructor. Creating an object using a constructor can be an expensive and lengthy operation. If you don't need to perform any initialization on an object, you can call the JNI function `AllocObject()` to create the object without invoking a constructor. This is especially useful if you are performing an operation within native code that requires a large number of objects being created, temporarily worked on, then destroyed.

`AllocObject()` simply takes a `jclass` value that specifies the class of object to create. The function returns a `jobject` that represents the new object instantiated from the given class. Since no constructors are executed, there is no additional setup time for locating method identifiers or invoking methods, which means you can create far more objects with far less overhead.

The Gotcha of Object Creation

The ability to manipulate objects from within native code is extremely useful and provides the means for interacting with the JVM in an extremely optimized and high-performance manner. However, there is a major gotcha to all these shenanigans: garbage collection. As you may have noticed, there are no corollary object deletion or deallocation methods defined within the JNI. Thus, you cannot arbitrarily delete Java objects. If you allocate or construct ten thousand objects within a native method, you are totally at the mercy of the garbage collector to deallocate the memory consumed by these objects.

This is actually no different from the way things work in Java code, but you may have mistakenly thought that you had more control over the matter by using native code. This is not the case. It is still a bad idea to wantonly create Java objects that you do not necessarily use to their full extent. The benefit, however, is that you have more direct control over how you interact with these objects after you have created them.

Notes

1. Of course, with the advent of 64-bit architectures, you may need to use a `long` value for pointer storage instead of an `int`.
2. Nonvirtual refers to the fact that, by default, all methods in Java are virtual methods.
3. Recall that Java constructors are declared with no return type, not even `void`.

Chapter 9. Classes and Objects with the RNI

Java programs revolve around classes and objects. So far, the RNI examples we've seen have manipulated data passed into the native methods via arguments. This makes sense for certains kinds of data, but there many other situations where we need to access data stored within a Java class or object. Furthermore, there are times when we need to invoke a Java method from native code or create an entirely new Java object within a native method and return the object to Java.

All of these operations imply some sort of knowledge about the Java class we are manipulating. When an object is passed into a native method as an argument, knowing this class information is not strictly necessary, since we can directly reference the fields of the object via the structure defined in the relevant include file. However, if that information is not available directly, we must be able to retrieve it indirectly by some means. This chapter discusses how to get that information with the RNI. In addition, it covers common object actions, such as retrieving and setting values of fields defined within a Java class, invoking methods defined within a Java class, and creating new Java objects.

Locating Classes

With the RNI, information representing a specific class is encapsulated in a structure called `ClassClass`. Most RNI functions that require class information, such as the functions that create objects and invoke methods, take a pointer to one of these structures, `ClassClass *`. Before you can use a pointer to class information to create an object or invoke a method, however, you must first locate the class information.

The RNI defines two functions for locating class information: `FindClass()` and `FindClassEx()`. Both functions take the name of a class and return a pointer to the appropriate `ClassClass` structure. The way in which they operate, however, differs slightly.

`FindClass()` is the more commonly used function. When given the fully-qualified name of a class, it either return the `ClassClass` information for that class or `NULL` if the class cannot be located. The following code fragment shows the use of `FindClass()` to locate the `java.lang.Object` class:

```
/* Attempt to locate the class information for java.lang.Object */
ClassClass *jlOClassblock = FindClass( NULL, "java/lang/Object", TRUE );

/* Test to see if the call was successful */
if ( jlOClassblock == NULL ) {
    fprintf( stderr, "Failed to locate class information!\n" );
  } else {
    fprintf( stderr, "Located class information!\n" );
  }
```

The first and last arguments to `FindClass()` are unimportant—always pass the value `NULL` for the first argument and the value `TRUE` for the third argument.

One thing you need to know about `FindClass()` is that it always causes the class to be loaded into the JVM if it hasn't already been loaded. Thus, if the class you are trying to locate has a static initializer block, the code in that block is executed when the class is loaded into the JVM. This might not be a desired side-effect, especially if the static initializer triggers additional code like native library loading.

To circumvent this potential problem, Microsoft has added the `FindClassEx()` function, which gives you more control over how locating classes interacts with the JVM's class loading mechanism. It also allows you to perform non-standard, Microsoft-specific operations, such as case-insensitive class searching.

If you want to prevent static initializers from being executed, you can pass the value `FINDCLASSEX_NOINIT` to `FindClassEx()` along with the class name. Similarly, to perform case-insensitive class searches, you can use the value `FINDCLASSEX_IGNORECASE`. Note that case-insensitive searches contravene the Java Language Specification for class naming—class names in Java *are* case-sensitive. Here is a code fragment that shows the use of optional arguments with `FindClassEx()`:

```
/* The class information */
ClassClass *jlOClassblock;

/*
 * Attempt to locate the class information for java.lang.Object
 * without invoking any static initializers
 */
jlOClassblock = FindClassEx( "java/lang/Object", FINDCLASSEX_NOINIT );

/* Test to see if the call was successful */
if ( jlOClassblock == NULL ) {
    fprintf( stderr, "Failed to locate class information!\n" );
    fprintf( stderr, "Trying case-insensitive search..." );

    /* Try a case-insensitive search for java.lang.object */
    jlOClassBlock = FindClassEx( "java/lang/object",
                                 FINDCLASSEX_IGNORECASE );

    if ( jlOClassblock == NULL ) {
        fprintf( stderr, "failed\n" );
      } else {
        fprintf( stderr, "succeeded!\n" );
      }
  } else {
    fprintf( stderr, "Located class information!\n" );
  }
```

The arguments for the operation mode of `FindClassEx()` can also be combined with a logical OR:

```
ClassClass *cb = FindClassEx( "someClass",
                              FINDCLASSEX_NOINIT | FINDCLASS_IGNORECASE );
```

Newer versions of the Microsoft JVM have added an extra RNI function that can be used to locate classes via a custom `ClassLoader`, as opposed to the default `ClassLoader`. This function, `FindClassFromClass()`, works like `FindClassEx()`, except that it takes an additional parameter of type `ClassClass *` that represents the custom `ClassLoader`.

The `FindClass()` and `FindClassEx()` functions locate classes using the JVM's class path, while `FindClassFromClass()` requires a custom `ClassLoader`. Sometimes, however, you want to look outside of the class path without going to the hassle of creating a custom `ClassLoader`. For just this situation, the RNI defines a function called `AddPathClassSource()`. It allows you to dynamically append or prepend new directories to the JVM's internal class path. This function takes two arguments: the path to add to the class path and a flag that indicates whether the path is to be appended or prepended. For example:

```
/* Try and locate the SymbolStone class in the default class path */
ClassClass *symbolStoneClassblock = FindClass( NULL, "SymbolStone", TRUE );
if ( symbolStoneClassblock == NULL ) {
    /* Haven't found the classblock, so append a develop-
ment path and retry */
    AddPathClassSource( "\dev\classes", TRUE );
    symbolStoneClassblock = FindClass( NULL, "SymbolStone", TRUE );
    ...
  }
```

One final technique that you can use to get the class information for a given class is to query an existing object of that class for its class information. This technique is particularly useful when you don't have the actual class name, but you have an object of the class. The RNI function `Object_GetClass()` takes an object and returns the class information in the form of a `ClassClass *`, as shown in the following code fragment:

```
RNIEXPORT void RNICALL
someClass_someMethod( struct HSymbolStone *stone ) {

    /* The class information for the SymbolStone class */
    ClassClass *symbolStoneClassblock = Object_GetClass( stone );
```

```
/* Check the class information is OK... */
if ( symbolStoneClassblock == NULL ) {
    fprintf( stderr, "Cannot extract class information from object!\n" );
  } else {
    fprintf( stderr, "Extracted class information!\n" );
  }

...
```

Testing Object and Class Characteristics

Java contains some useful keywords that can be used to test certain characteristics of objects and classes such as `instanceof`. The RNI provides native code mirrors of these functions affording you considerable power in class manipulation within your native methods.

The RNI defines two forms of `instanceof` which do similar, but not quite identical, things. They are also both rather confusingly named similarly.

The first of these two functions is called `isInstanceOf()` and returns a boolean value indicating whether or not a given Java object is instantiated from the class named in the second argument. For example, the following code stub tests whether or not a given object was instantiated from the `SymbolStone` class.

```
/** The object of unknown instantiation */
HObject *object;

/** Test whether this is an instantation from the SymbolStone class */
if ( isInstanceOf( object, "SymbolStone" ) == TRUE ) {
    fprintf( stderr, "The object is of class SymbolStone\n" );
  } else {
    fprintf( stderr, "The object isn’t of class SymbolStone\n" );
  }
```

The second form of `instanceof` defined within the RNI is the function `is_instance_of()` which again returns a boolean value. This function tests whether or not the given object can be cast to the given class, specified by a class information structure.

For example, the following code shows how you can test whether or not an object of class `SymbolStone` can be safely cast to `java.lang.String`:

```
/** The symbol stone object to try casting */
```

```
    HObject *stone;

    /** Locate the class information for java.lang.String */
    ClassClass *jlsClassblock = FindClass( NULL, "java.lang.String", TRUE );

    /** Test to see if the object can be safely cast */
    if ( is_instance_of( stone, jlsClassblock, NULL ) == TRUE ) {
        fprintf( stderr, "SymbolStone ob-
ject can be cast to java.lang.String!\n" );
      } else {
        fprintf( stderr, "SymbolStone object can-
not be cast to java.lang.String!\n" );
      }
```

A further useful function defined within the RNI is `is_subclass_of()` which determines whether or not a given class is a subclass of another class. This function takes two class information pointers as arguments and returns a boolean value indicating whether or not the subclass relationship is present between the two classes. For example, the following code stub tests whether or not `java.lang.String` is a subclass of `java.lang.Object`.

```
    /** The class information for java.lang.String */
    ClassClass *jlsClassblock = FindClass( NULL, "java.lang.String", TRUE );

    /** The class information for java.lang.Object */
    ClassClass *jloClassblock = FindClass( NULL, "java.lang.Object", TRUE );

    if ( is_subclass_of( jlsClassblock, jloClassblock, NULL ) == TRUE ) {
        fprintf( stderr, "java.lang.String is a subclass!\n" );
      } else {
        fprintf( stderr, "java.lang.String is not a subclass!\n" );
      }
```

There are several more functions defined within the RNI that allow testing and fetching of various aspects of classes. These generally are RNI reflections of the methods defined within the `java.lang.Class` class and can be summarised by the following table.

Table 9-1. `java.lang.Class` to RNI Functions

`java.lang.Class`	Corresponding RNI Function
`getModifiers()`	`Class_GetAttributes`
`getField()`	`Class_GetField()`

`java.lang.Class`	**Corresponding RNI Function**
`getInterface()`	`Class_GetInterface()`
`getMethod()`	`Class_GetMethod()`
`getName()`	`Class_GetName()`
`getSuperclass()`	`Class_GetSuper()`

Accessing Fields

There are two operations that you may perform upon fields within a class. You may read the values currently set within Java objects or classes and, if you have modified these values within your native code, you can write the modified values back. The two operations are usually called "*getting*" and "*setting*".

Getting and setting is one of the more common things that you will wish to do. They enable you to pass data directly back into Java objects that can be then referenced within Java code. Similarly, it enables you to read data that has been created or calculated within Java code in your native methods.

For example, one of the favourite questions asked by neophyte native method programmers is "how can I store a C `struct` or C++ class within a Java object?". This is quite a common thing to want to do since our legacy code will undoubtedly create all sorts of objects and data structures containing information used internally by the legacy code. Ordinarily, our C- or C++-based program would track these using pointers or variables, but Java doesn't have pointers!

The neat trick here is to remember that a pointer is basically just a number representing a memory address. Therefore, it's easy to store pointers within a Java object in a variable declared as an `int`. Within your native code, if you wished to store a C `struct`, you could simply use `malloc()` to allocate the amount of memory that the `struct` takes up then set the value of the Java `int` field to that of the memory address at which the `struct` is located.

Similarly, in later operations when you need to recover your original data structure, you only need to extract the `int` value from the Java variable and cast it to being a pointer to the `struct`. And that's firstly, how you store C/C++ data structures within Java objects and secondly, one reason why manipulating Java field data within native code is incredibly useful!

Another extremely good reason why this functionality is useful is that you can update several different Java variables within a single native method instead of having to return values from the native method and setting the Java variables to the return values. This technique, although workable, only allows you to set a single value per native method which can provoke major architectural headaches.

Unlike the JNI, the RNI does not define actual functions for getting and setting fields within Java objects or classes. You may access the fields of a class directly *via* the C `struct` that represents the class as defined in the appropriate automatically generated `#include` file.

As objects within native code are referenced *via pointers*, accessing fields can be achieved by a simple dereference operation. For example, to access the `_pData` field within the `SymbolStone` class

```
/** Returns the name of the stone */
RNIEXPORT struct Hjava_lang_String * RNICALL
SymbolStone_getName( HSymbolStone *arg ) {
    /** Extract the pointer */
    SymbolStone_t *stone = (SymbolStone_t *)arg->_pData;

    ...
```

This illustrates that the fields contained within a Java object or class can be referenced and manipulated in exactly the same way as you would access standard C variables stored within a `struct`. You can check the `#include` file for the actual C data types that the Java variables are represented by.

Furthermore, if a variable stored within a Java object is actually another object, this can be further dereferenced and manipulated. For example, within the `java.awt.Component` class, there exists a variable called `minSize` of type `java.awt.Dimension` that holds the minimum size of the component. Therefore, to access the information stored within that particular variable, you would need to run `msjavah java.awt.Dimension` and peek into the structure of that object. If you do you can see that within that class, two variables exist called `width` and `height` which tell you the dimensions of the component.

In cases where multiple dereferencing is required, you must remember to generate an include file for that class and also include it explicitly within your code otherwise the structure of the object will not be available to you. The following code stub illustrates how to access the dimensions of the minimum size of a component.

```
#include <MyComponent.h>
#include <java_awt_Dimension.h>

RNIEXPORT void RNICALL
MyComponent_getMinimumSize( struct HMyComponent *component ) {

    /**
     * Dereference the “minSize” variable within the component and extract
     * the width value
     */
    int width = component->minSize->width;

    /**
     * Dereference the “minSize” variable within the component and extract
     * the width value
     */
    int height = component->minSize->height;
  }
```

Setting the values within objects is essentially an identical operation to simply referencing the values. The only difference between getting and settings within the RNI is that a new value of an appropriate data type is assigned to the referenced field. For example, to set the `_pData` field within a `SymbolStone` object, the following code fragment can be used.

```
    /** Create a new SymbolStone object */
    HSymbolStone *stone = ...;

    /** Set the _pData pointer... */
    stone->_pData = (long)sptr;

    ...
```

A final note on this topic concerns referencing the values stored within objects. In the above examples, I have shown the simple cases where variables have been mapped to standard C data types. What happens if the variable we are referencing is something like a string?

In these instances, the appropriate functions within the RNI should be used to manipulate these objects. For example, in the case of strings the standard string manipulation functions that were discussed in Section should be used to access and manipulate the value stored within it.

Setting the values of objects is equally simple. This might be useful if you have created a new Java object within native code and wish to replace an existing object with the new one. Similarly, you might wish to update the value contained within a string to a new value. Within the context of the RNI, this type of operation is essentially a pointer assignment which alters the internal references used by Java to access objects.

And that's all there is to field manipulation within the RNI! It's a far more succinct syntax than field manipulation within the JNI, but has more caveats in that you must always remember to generate and include the necessary `#include` files within your code if you are accessing multiple object types. Similarly, if you change the structure of your class in any way, you *must* regenerate the include file for that class otherwise the representation in memory of the object and the representation stored within the include file may not match. This is a bad thing since you are directly operating on raw memory addresses and can potentially corrupt or crash the JVM completely by overwriting the contents of the wrong memory address! Be careful!

A slightly less powerful, but more flexible, approach to getting and setting field data also is defined within the RNI. Within the Java class `java.lang.Class`, there exists a set of methods such as `getField()` that will retrieve *reflections* of the fields contained within a Java class. Each field is encapsulated by the `java.lang.reflect.Field` class which in turn contains methods such as `getInt()`, `getDouble()` and `setInt()` and `setDouble()`. A method exists for getting and setting field values of all the primitive datatypes and objects.

The RNI features a native code equivalent to these operations allowing you to programmatically set the contents of various fields using functions rather than by directly addressing the member variables of the C `struct` representing the class.

As I mentioned before, these getting and setting operations within Java operate on an instantiation of the `java.lang.reflect.Field` class which encapsulates all the information stored on a particular field. The RNI defines an equivalent data type that is used for all field-related functions. This data type is of type `struct fieldblock` and can be fetched from the class information for the correct class.

Once the class information for the relevant class has been successfully located, we can use the RNI function called `Class_GetField()` to return the `struct fieldblock` value encapsulating the desired field within the class. The following code stub demonstrates how this operation works.

```
    /** The class information for the SymbolStone class */
    ClassClass *symbolStoneClassblock = FindClass( NULL, "Symbol-
Stone", TRUE );

    /** Test that we’ve found the class information... */
    if ( symbolStoneClassblock != NULL ) {
        /** Locate the field information for the “_pData” field */
        struct fieldblock *pDataFieldInfo =
            Class_GetField( symbolStoneClassblock, "_pData" );

        /** Test that we’ve found the field info... */
        if ( pDataFieldInfo == NULL ) {
            fprintf( stderr, "Failed to locate field info within class!\n" );
          } else {
            fprintf( stderr, "Located field info within the class!\n" );
          }
      } else {
        fprintf( stderr, "Failed to locate the class information!\n" );
      }
```

Now that you have the correct information to reference any field within a Java object, you can use the getter and setter methods to manipulate the values stored within them. Each primitive data type and the `Object` data type all have getter and setter methods associated with them all following a standard declaration and can all be used in the same way. You *must* ensure that you use the correct function for the correct data type otherwise unpredictable results might occur resulting in your program, the JVM or both crashing.

This technique is certainly more verbose than directly manipulating the member variables of the `struct` representing the Java class in question, but it makes your code somewhat more readable and does ensure a certain level of type-safety not afforded by the more direct approach.

It does essentially boil down to a matter of personal preference in the long run. My personal preference is to use the direct approach since it is considerably less wordy. However, if your code did use the information for each field in more detail, for example, you were developing a classfile browser or IDE, you might find the extra information to be worth the effort of writing more verbose code.

Invoking Methods

In object-orientated languages such as Java, invocation of methods declared and implemented within classes is an extremely important concept. For example, variables within a class or object should not be accessed directly but *via accessor functions*. Similarly, the notion of *method overriding* in subclasses means that complex chains of method invocation might occur.

For example, our example program might create a `SymbolStone` object. This object, in turn, might contain an array of `String`s representing the alternative names the symbol stone is known by. If you wished to print out information on the stone, you might invoke the `SymbolStone.toString()` which, in turn, would iterate through the array of alternative names further invoking the `toString()` method on those.

Given that invoking methods is a fairly useful thing to do it would stand to reason to suggest that being able to do so from native code would also be useful. This functionality is most useful when you have Java code that carries out a defined task in a modular fashion and you need to be able to perform this operation during the running of a native method. For example, a rendering package might execute some native code to raytrace an image. After each line of the image has been rendered, it might be desirable to update a progress bar AWT component in the GUI to inform the user of the program what the progress of the rendering is.

In this instance, all the functionality for manipulating a progress bar component lies within Java code. Therefore, by invoking appropriate Java methods from native code, you would be able to continue to use the progress bar from native code without having to rewrite functionality that already works perfectly well.

Specifying Methods

Before you can invoke Java methods from within native code, you must uniquely identify the method that you are interested in. This is done by a composite of the *class* from which an object is instantiated, the *name* of the method that you wish to execute and the *signature* of the method.

The first two parameters are fairly obvious in identifying methods but what has a type signature got to do with it? The answer lies in one of Java's more powerful features called *method overloading*. This allows you to define multiple methods with the same name and return type[1] but different parameters. For example, a 3d graphics library might have a class called `Point` that defines a point in 2- or 3-dimensional space. This class also contains methods that let you specify the point itself. The overloaded method declarations for this method would look like

```
void vertex( int x, int y )
void vertex( float x, float y )
void vertex( int x, int y, int z )
void vertex( float x, float y, float z )
```

Therefore, if you simply attempted to locate a method by its name and class only, you have 4 potential results in the case of the `vertex()` method. This is not exactly unique! Specifying the type signature for the desired method will ensure that you are returned the method identifier for the correct method.

For example, the type signatures for the above methods respectively work out as

```
(II)V
(FF)V
(III)V
(FFF)V
```

The technique used to assemble the type signature is described in full in Chapter .

Invoking The Located Methods

There are two main ways defined in the RNI to invoke Java methods, the slower but easier way and the faster but harder way. The easy and slower way hides some additional internal method handling that you might not want to deal with. However, if you are optimizing your code for maximum performance, using the underlying internal methods instead of the more convenient methods can be very beneficial.

The easy way to invoke Java methods from native code is to use the RNI functions `execute_java_dynamic_method()` for instance methods or `execute_java_static_method()` for statically declared methods. These functions take the object or class against which you wish to invoke the method, the method identification parameters and any arguments to the method as parameters to the call. The parameters for both instance and static methods are identical.

Using either of these methods is very straightforward and needs little explanation. For executing the `vertex()` instance methods from the example shown above, you could simply write

```
execute_java_dynamic_method( NULL, aPoint, "ver-
tex", "(FF)V", 111.11, 222.22 );
```

where `aPoint` is an instance of the `Point` class. Each required argument for the Java method is simply listed after the type signature for the method.

Class methods, that is methods declared as being `static` within Java, require an additional stage of preparation. This extra stage is necessary because you need to pass information on the class in which the method is declared rather than an object. Therefore, you need to find that information out in advance.

Fortunately, this is easy to do and can be effected by using either `FindClass()`, `FindClassEx()` or `Object_GetClass()` all of which are discussed in the previous section. Once you have located the class information for the desired class, you can now invoke your Java method from native code using `execute_java_static_method()` with the following code snippet.

```
    /** Locate the class information for the Point class */
    ClassClass *PointClassblock = FindClass( NULL, "Point", TRUE );

    /** Execute the static method “vertex(float,float)” */
    execute_java_static_method( NULL, PointClassblock, "vertex", "(FF)V",
                                111.11, 222.22 );
```

These forms of method invocation can be quite expensive in terms of performance. For each invocation of either of these RNI functions, a lookup is done for *method information* on the relevant method internally. This additional lookup is actually unnecessary in cases of multiple executions of the same method. The following psuedo-code illustrates the idea more clearly.

```
    /** Locate the class information for the Point class */
    Class *PointClassblock = FindClass( NULL, "Point", TRUE );

    /** Specify a million points */
    for ( int i = 0 ; i < 1000000 ; i++ ) {
        // execute_java_static_method( NULL, PointClassblock, "vertex",
        //                             "(FF)V", 111.11, 222.22 );
        /** Locate the method information for the vertex() method */

        /** Execute the method using the fetch method information */
      }
```

As can be seen, by using these more convenient method invocation functions, we might be locating method information unnecessarily 999,999 times in the main loop! Can we improve on this at all?

Fortunately, yes we can, and this is the slightly more involved, but much faster, way of invoking methods. This is a more low-level approach to method invocation and requires an extra stage of processing before you can invoke your methods.

The basis of this technique is to *manually* locate the *methodblock* for the method prior to invoking it. This is a chunk of data that represents the Java method itself. The performance boost comes with the fact that you can reuse the methodblock over and over again for subsequent invocations of that method.

Java methods are invoked using this technique with the `do_execute_java_method()` RNI function which can handle the invocation of both instance and static methods by passing appropriate values as an argument.

The methodblock itself is located using the `get_methodblock()` which, if successful, will return a pointer to a structure of type `struct methodblock`. The example shown above could be re-written to use the fast method invocation technique in the following way.

```
struct methodblock *vertexMethodblock =
    get_methodblock( aPoint, "vertex", "(FF)V" );
do_execute_java_method( NULL, aPoint, NULL, NULL, vertexMethodblock, FALSE,
                        111.11, 222.22 );
```

The parameters of this method require a little explanation. Parameter 1 is always `NULL` as are parameters 3 and 4. Parameter 2 is either the object or classblock you wish to invoke the method against depending on whether the method is instance- or class-level. Parameter 5 is the methodblock for the method that you have already located. The final formal parameter is a boolean value signifying whether or not the method is static or not. In this example, the method is instance-level which implies the parameter should be `FALSE`. In cases of class methods, this parameter should be `TRUE`. After these parameters, the actual parameters to be supplied to the method itself should be specified.

The fast method of invoking methods is best used where repeated invocations of the same method is required within a native method, within a tight loop. For example, our million vertex loop example from above could be re-written in the following way which is appreciably faster than the original code.

```
    /** Locate the method information for the vertex() method */
    struct methodblock *vertexMethodblock =
        get_methodblock( aPoint, "vertex", "(FF)V" );

    /** Specify a million points */
    for ( int i = 0 ; i < 1000000 ; i++ ) {
        /** Execute the method with the pre-fetched methodblock */
        do_execute_java_method( NULL, aPoint, NULL, NULL, vertexMethodblock,
                                FALSE, 111.11, 222.22 );
      }
```

The saving of not having to locate the methodblock will increase relative to the number of iterations of the loop. However, in cases where a single method invocation is desired, using the `execute_java_dynamic_method()` and `execute_java_static_method()` functions is probably easier to use and takes the same amount of time from a performance point of view.

As with many aspects of native method programming, "profile before you optimize". Using the slower, more convenient, methods may well give you perfectly acceptable performance for your application without having to optimize each and every method invocation.

In newer versions of the RNI Specification, additional functions relating to method invocation have been added given you further control over what activities you can carry out from native methods.

A set of functions relating to interface-based method execution have been added allowing you to invoke a method against an object through an interface which that object implements. That is, if the `SymbolStone` class implemented the `Runnable` interface, you can now execute the `run()` interface method from native code.

The main function which defines this functionality is `execute_java_interface_method()` which operates in a similar way to `execute_java_dynamic_method()`. The main difference between these two functions is that since the method is being invoked within a class different to that from which the object was instantiated, an additional argument containing the class information of the interface is required.

For example, if we used the example of the `SymbolStone` class implementing the `Runnable` interface, the invocation of the `run()` method can be expressed in the following way.

```
/** Locate the class information for the Runnable interface */
ClassClass *runnableClassblock =
    FindClass( NULL, "java/lang/Runnable", TRUE );

/**
 * Invoke the “run()” method in the interface against the
 * symbol stone
 */
execute_java_interface_method( NULL, stone, runnableClassblock,
                               "run", "()V" );
```

Unfortunately, the `do_execute_java_method()` function cannot be used to speed up the execution of interface methods at all in the current version of the RNI Specification. Therefore, repeated execution of interface methods may begin to cause bottlenecks within your native methods.

The RNI Specification added some other functions to the topic of method invocation with the addition of `execute_java_dynamic_method64()`, `execute_java_static_method64()` and `execute_java_interface_method64()` which now correctly cope with methods that return a true 64-bit value from methods. This is particularly useful when a method returns a `long` value and ensures that the value is not truncated to suit the processor architecture upon which you are running the JVM. This typically occurs on platforms where the size of a `long int` value is less than 64 bits.

The final new functions for method invocation that have been added to the RNI Specification simply specify a different way of passing arguments to Java methods. With `execute_java_dynamic_method()` and friends, the arguments required by the target Java method are simply passed as a list of individual values. The `execute_java_dynamic_methodV()`, `execute_java_static_methodV()` and `execute_java_interface_methodV()` methods act in a similar way but the required arguments are stored within an ANSI varargs list.

Creating Java Objects

In object-orientated languages, such as Java, you will be using objects on an extremely regular basis. Objects, or instantiations of classes, form the basic building blocks that your programs use to manipulate data. For example, if you wish to perform operations on a string, that string is encapsulated by the class `java.lang.String`. Similarly, our example system encapsulates the notion of a symbol stone in the `SymbolStone` class.

The creation of Java objects within the average Java program is an extremely frequent operation be they objects being created by the JVM in order to carry out tasks underneath the bonnet or objects created by you in order to execute the logic of your program. It would also be extremely useful to be able to create objects from within native methods also.

An example of where the ability to create object from within native methods is where legacy code creates a new object internally that needs to be *reflected* in Java. To return all the bits of data that the object comprises of from native code to Java is unwieldy and impractical. However, creating a new object within native code and returning an `HObject` value for that object is elegant and fast.

Another subject touched on earlier in an earlier chapter is the allocation of arrays of objects from within native code. The RNI function to allocate arrays of objects from within only allocates the space for the array and does not actually initialize each member element. Therefore, to populate this fledgling array from native code, we must be able to create Java objects natively.

The RNI defines a single function for object creation from within native code, `execute_java_constructor()`. This function operates in a similar way to the method execution functions since the *class* that you wish to construct an object from is a required parameter. The other extremely important parameter is the *signature* of the constructor that you wish to invoke.

Constructors are actually just methods with special meaning. Therefore, the technique used to construct a signature for a method is equally pertinent for constructors, the only difference being that constructors do not have a return type at all[2]. The remaining arguments are just the arguments that the constructor requires.

To illustrate the object creation process from native code, we can look at the full implementation of the `SymbolStone.load()` method:

```
/**
 * Scans the given data file and instantiates an array of SymbolStone objects
 * from it
 */
RNIEXPORT HArrayOfObject * RNICALL
SymbolStone_load( ClassSymbolStone *arg,
                  struct Hjava_lang_String *filename ) {

    /** The return array of objects... */
    HArrayOfObject *rv = NULL;

    /** Temporary buffers */
    char filenameChars[1024];

    /** Extract the filename */
    javaString2CString( filename, filenameChars, sizeof( filenameChars ) );

    /** Load the data from file into C structures... */
    int numRecords = 0;
    SymbolStone_t *stones = load( filenameChars, &numRecords );
    SymbolStone_t *sptr = stones;

    /** Safety check... */
    if ( stones == NULL || numRecords == 0 ) {
        fprintf( stderr, "stones array == NULL or %d records == 0\n", num-
Records );
        return NULL;
      }

    /** Allocate an array of jobjects... */
    rv = (HArrayOfObject *)ClassArrayAlloc( T_CLASS, numRecords, "Symbol-
Stone" );
    if ( rv == NULL ) {
        fprintf( stderr, "Failed to allocate return array of SymbolStone ob-
jects\n" );
        return NULL;
      } else {
        fprintf( stderr, "Allocated array of %d records\n", numRecords );
      }

    for ( int i = 0 ; i < numRecords ; i++ ) {
        HSymbolStone *stone =
            (HSymbolStone *)execute_java_constructor( NULL, "Symbol-
Stone", NULL, "()" );
        if ( stone == NULL ) {
            fprintf( stderr, "failed to construct new instance of Symbol-
Stone class\n" );
            return NULL;
          }

        /** Set the pData pointer... */
        stone->_pData = (long)sptr;

        /** Stuff the object into the array */
        rv->body[i] = (HObject *)stone;

        /** Increment the pointer... */
        sptr++;
      }

    return rv;
  }
```

The parameters for `execute_java_constructor()` are worth some explanation. Parameter 1 is always `NULL`. Parameter 2 is the fully-qualified name of the class including package. The package delimeter to be used a forward slash (`/`). For example, if the `SymbolStone` class belonged to the `com.oreilly` package, the fully-qualified class name is `com/oreilly/SymbolStone`. Parameter 3 is also always `NULL` and parameter 4 is the type signature for the constructor you wish to invoke. This ensures the correct constructor is always invoked. After these parameters have been specified, you should list the appropriate parameters required by the constructor in question.

The value returned from executing `execute_java_constructor()` is of type `HObject`. This data type can be cast to any class type whatsoever, *e.g.*, you can cast it from `HObject` to `HSymbolStone` quite happily in cases where a new `SymbolStone` object has been created and `HObject` is simply an opaque Java object reference just as all Java objects can be cast to `java.lang.Object`.

Similarly, if an error occurs during object instantiation, the value `NULL` will be returned and an exception will be thrown further specifying what the problem was.

Newer versions of the Microsoft JVM have added another couple of RNI functions which allow you to optimize the invocation of Java constructors in a way similar to the way in which we can optimize Java method invocations.

`execute_java_constructor()` operates by taking the name of the class to execute a constructor of and the type signature of the appropriate Java constructor. While this approach is quite simple to use, from an internal point of view, it can be quite costly.

This is due to the fact that a Java constructor is treated as being a method of name `<init>` with a return type of a Java object of the relevant class. Therefore, for every call to `execute_java_constructor()`, the JVM needs to look up the methodblock for the relevant constructor and invoke it. Over a number of iterations, the repeated, and unnecessary, methodblock lookups starts to mount up in terms of performance overhead.

The new RNI functions that have been added to allow for repeated constructor invocation with little overhead are called `execute_java_constructor_method()` and `execute_java_constructor_methodV()` which operate in a similar way to `do_execute_java_method()` and `do_execute_java_methodV()`.

This allows us to optimize Java constructor invocation by looking up the methodblock representing the desired constructor *once*, then making repeated calls to `execute_java_constructor_method()` passing that methodblock as an argument.

For example, the example listed above showing the construction of individual `SymbolStone` objects to be used as array elements could be re-written in a more optimized fashion as follows.

```
    ...

    /** Locate the methodblock for the desired SymbolStone constructor */
    struct methodblock *symbolStoneCtorMethodblock =
        Class_GetMethod( FindClass( NULL, "SymbolStone", TRUE ),
                         "<init>", "()" );

    /** Now, populate the array by creating new stones for each element */
    for ( int i = 0 ; i < numRecords ; i++ ) {
        HSymbolStone *stone =
            (HSymbolStone *)execute_java_constructor_method( symbolStoneC-
torMethodblock );
        if ( stone == NULL ) {
            fprintf( stderr, "failed to construct new instance of Symbol-
Stone class\n" );
            return NULL;
          }

        ...
      }

    ...
```

Therefore, the RNI can be used in both simple and complex ways to maximize either your ease of programming or your application performance when allocating Java objects from native code.

Notes

1. And similar functionality if your code is written correctly and clearly.
2. An extremely common mistake is to specify that the constructor either returns a value that is the type of the class or `void`.

Chapter 10. Exception Handling

Java implements its error handling capabilities with exceptions. An exception is a Java object that is “thrown” when a particular error occurs. Once an exception is thrown, it propagates through the Java method invocation stack until a method “catches” it and deals with it or it reaches the top of the invocation stack and the program execution stops.

Exceptions are a significant improvement over the simple return values returned by functions in C. Since an exception is an actual object, instead of a simple value, it can contain more information about the exact reason that a particular method failed. There are different exception types for different kinds of errors, such as `ArrayIndexOutOfBoundsException` for trying to go off the end of an array and `NullPointerException` for trying to access a reference that is set to `NULL`.

The JNI and RNI define several functions that allow you to interact directly with exceptions, in terms of both throwing new exceptions from native code and also catching and handling them in native code.

Throwing Exceptions

If the legacy code you are interfacing with is well-written, there is a good chance that each function has a set of defined return codes. Similarly, if you are writing new native code, you probably have a defined set of return values for each of your important functions. While it is certainly possible to define your native methods to return these error values back to Java for processing, but it is far more sophisticated and extensible to throw exceptions from your native code that can be caught within Java.

Throwing Exceptions with JNI

The JNI defines several functions that allow you to throw an exception from native code: `ThrowNew()`, which creates a new exception object for you and throws it, `Throw()`, which throws a previously constructed exception object, and `FatalError()`, which immediately halts the JVM. `ThrowNew()` is the most useful of these functions, since it handles creating the exception object for you. for you. Here is the function prototype for `ThrowNew()`:

```
jint ThrowNew( JNIEnv *env, jclass clazz, const char *message )
```

Calling the `ThrowNew()` function is exactly like the following Java code you might use to throw a new `NullPointerException` object:

```
throw new NullPointerException( "Yikes! A NULL pointer!" );
```

The only real difference between throwing an exception in Java and in native code is that in native code we need to provide a `jclass` object that represents the class of the exception to throw. In this case, to throw a `NullPointerException`, we need the `jclass` value for the `java.lang.NullPointerException` class, which we can retrieve with `FindClass()`. Thus, we can throw a `NullPointerException` from native code as follows:

```
/* Throw a NullPointerException */
env->ThrowNew( env->FindClass( "java/lang/NullPointerException" ),
               "Yikes! A NULL pointer!" );

/* Return to Java immediately! */
return;
```

When you make the call to `ThrowNew()`, the exception isn't actually thrown, but instead placed on a pending stack. The actual throwing process is triggered by returning from the method in which the exception was made pending, which in this case is the native method. Therefore, it's good practice to immediately return control from the native method to Java after you call `ThrowNew()`, to ensure that the exception is thrown as soon as possible.

To illustrate the throwing of exceptions from within native code, we'll look at the `describe()` method of the `SymbolStone` class in our example application. This method simply returns a string describing the stone based on its symbols. If the stone has no symbols, a `NoSymbolsException` is thrown. The declaration of the native method is written in the usual way with the addition of the `throws` clause declaring which exceptions might be thrown during the native method execution.

```
/** Attempts to describe the stone... */
public native String describe() throws NoSymbolsException;
```

```
/** Describes the stone... */
JNIEXPORT jstring JNICALL
Java_SymbolStone_describe( JNIEnv *env, jobject arg ) {
    char rv[1024];

    /** Extract the pointer */
    jfieldID pDataFieldID = \
        env->GetFieldID( env->GetObjectClass( arg ), "_pData", "I" ); \
    SymbolStone_t *stone = \
        (SymbolStone_t *)env->GetIntField( arg, pDataFieldID );

    /** Safety check */
    if ( stone == NULL || (int)stone == -1 ) {
        return JNI_FALSE;
```

```
      }

    if ( stone->numSymbols == 0 ) {
        sprintf( rv, "%s has no symbols!", stone->name );
        env->ThrowNew( env->FindClass( "NoSymbolsException" ), rv );
        return NULL;
      } else {
        sprintf( rv, "%s has %d symbols", stone->name, stone->numSymbols );
        return env->NewStringUTF( rv );
      }
  }
```

The `Throw()` function performs exactly the same operation as `ThrowNew()`, but it requires you to construct an exception of the correct class manually and pass that to `Throw()`. I personally recommend staying away from this function, as it offers up plenty of scope for incorrectly constructing exception objects, not to mention that it involves far more work on your part for exactly the same effect.

The final way you can signal that an error has occurred within your native code is to use the `FatalError()` function. This function is not to be taken lightly, as it flags a fatal error within the JVM, which causes it to shutdown immediately and report the specified message as the reason as for the shutdown. `FatalError()` executes *immediately* (i.e., it isn't put made pending), and it is not interruptable nor can it fail.

The decision to invoke `FatalError()` should only be taken after considering the consequences of doing so. If you have multiple programs running within a browser environment, such as Netscape Navigator, bear in mind that although there are separate programs in operation, only *one* JVM is likely to be running. Thus, if you decide to invoke `FatalError()` in a Java program, all of the programs running within the browser will die immediately. In addition, the browser will probably require restarting, as the JVM has now aborted and will not be able to run any future applets.

Throwing Exceptions with RNI

The RNI defines a function called `SignalError()` that is similar to JNI's `ThrowNew()`. `SignalError()` takes the class name of the exception object to throw and the message to be associated with the new exception, creates a new exception object of the desired class, and makes it pending. Here's how to throw a `NullPointerException` with RNI:

```
/* Throw a NullPointerException */
SignalError( NULL, "java/lang/NullPointerException",
             "Yikes! A NULL pointer!" );

/* Return to Java immediately! */
```

```
return;
```

Just as with the JNI, the exception is placed on a pending stack and not actually thrown until the method returns, so your code should return immediately.

The RNI also defines a variant of `SignalError()` called `SignalErrorPrintf()`. This function performs exactly the same action as `SignalError()`, but instead of passing a single string that contains the descriptive message, you can pass arguments in the form that `printf()` uses. For example, if you want to throw an exception that lists not only the error message, but also the line number and C source file the exception was thrown in, you can use the following call to `SignalErrorPrintf()`:

```
/* Throw a NullPointerException */
SignalErrorPrintf( "java/lang/NullPointerException",
                   "Yikes! A NULL pointer occurred in %s, line %d",
                   __FILE__, __LINE__ );

/* Return to Java immediately! */
return;
```

As with `printf()`, the number of arguments specified after the format string should match the number of variables specified within the format string.

With `SignalErrorPrintf()` we can now implement an RNI version of the `SymbolStone.describe()` native method that may throw a `NoSymbolsException`.

```
/** Describes the stone... */
RNIEXPORT struct Hjava_lang_String * RNICALL
SymbolStone_describe( HSymbolStone *arg ) {

    /** Extract the pointer */
    SymbolStone_t *stone = (SymbolStone_t *)arg->_pData;

    /** Safety check */
    if ( stone == NULL || (int)stone == -1 ) {
        return NULL;
      }

    if ( stone->numSymbols == 0 ) {
        SignalErrorPrintf( "NoSymbolsException",
                           "%s has no symbols", stone->name );
        return NULL;
      } else {
        char rv[1024];
        sprintf( rv, "%s has %d symbols", stone->name, stone->numSymbols );
        return makeJavaString( rv, strlen( rv ) );
      }
  }
```

The RNI also provides the `exceptionSet()` function to throw a pre-created exception object, just like with the JNI `Throw()` function. `exceptionSet()` is a lot more convoluted to use than either `SignalError()` or `SignalErrorPrintf()`, so I recommend avoiding it in favor of the simpler methods. Finally, the RNI has no corresponding function for JNI's `FatalError()` function.

Catching Exceptions

It is impossible to implement a native code equivalent of Java's `try`/`catch` statements, in that you cannot use native code to catch an exception that has been thrown within a Java class. However, you can catch exceptions that have been thrown as a result of the failure of a JNI or RNI function call.

Consider, for example, the `FindClass()` function, which returns a non-`NULL` value (i.e., `jclass` in JNI, `ClassClass` in RNI) when it successfully locates the class you are looking for. When the function fails, however, it returns `NULL`. This failure could be for a number of reasons, so the function also throws an exception to provide more useful information. These exceptions include `ClassFormatError`, `ClassCircularityError`, and `NoClassDefFoundError`, relating to different problems with the class you have tried to locate.

Here's some Java code that corresponds to a call to `FindClass()`:

```
Class someClass = Class.forName( "NonExistentClass" );
```

With this Java code, you can wrap the call within a `try`/`catch` statement to catch the various exceptions as soon as one is thrown. Within native code, however, all you know is that `FindClass()` has failed because it has returned `NULL`. How exactly can we deal with the exception that's now waiting to be thrown?

We've already seen the first technique for dealing with exceptions, when we looked at throwing exceptions from native code. This technique involves declaring the native method that calls `FindClass()` as one that can throw the various exceptions that can be generated by the function. Then, in the native code, we can simply return immediately after a failed call to `FindClass()` and allow the Java code to deal with the exception.

This technique is perfectly legitimate: the operation failed and Java can be used to make a recovery from the situation. There are cases, however, when you might want to ignore certain exceptions or do something else to handle it without leaving the native method. This requires some ability to manipulate exceptions within native code.

For example, the `SetObjectArrayElement()` JNI function can result in a couple of different exception objects being thrown, namely `ArrayIndexOutOfBoundsException` and `ArrayStoreException`. These are both fairly important errors, but `ArrayIndexOutOfBoundsException` is one that we could recover from by simply recalculating another array index at which to place the object.

Within native code, there are really only three operations we can perform to handle an exception: detect that an exception has occurred; clear any pending exceptions; and print a stack trace showing where the exception occurred.

The first of these operations is the native code equivalent to the Java `try`/`catch` construct. With the JNI, we can call the `ExceptionOccurred()` function after invoking *any* function that might throw an exception. If an exception has been thrown, `ExceptionOccurred()` returns a `jthrowable` value that corresponds to the `Exception` object that is pending. Otherwise, if there are no pending exceptions, `ExceptionOccurred()` returns `NULL`.

To provide identical functionality to `try`/`catch` in native code, we need to do a little more work, however. The `jthrowable` value simply refers to a general exception object, so to provide error handling based on the exception type, we need to work out which class the exception belongs to. This is simply a matter of locating the class information for a class and comparing it against the class of the `jthrowable` returned by `ExceptionOccurred()`. For example, here's a little piece of code which handles recovery from an `ArrayIndexOutOfBoundsException`:

```
    jthrowable exception;
    int index;
    jobject someObject;
    jobjectArray anArray;

    ...

    env->SetObjectArrayElement( anArray, index, someObject );
    if ( ( exception = env->ExceptionOccurred() ) != NULL ) {
        /* The class information for the ArrayIndexOutOfBoundsExcep-
tion class */
        jclass aioobClassblock =
            env->FindClass( "java/lang/ArrayIndexOutOfBoundsException" );

        /* Compare the class of the thrown exception to the AIOOBException */
        if ( env->GetObjectClass( exception ) == aioobClassblock ) {
            /* Clear up the exception */
            env->ExceptionClear();
            /* Execute the fallback code here.... */
          } else {
            /* Return immediately and let the excep-
tion be dealt with in Java */
            return;
          }
      }

    ...
```

As you can see, this technique is quite wordy and it only gets worse as you need to handle multiple exception types! I recommend using it sparingly, although it is a very powerful way of providing more robust native methods.

A further caveat is that the RNI does not support the detection of the class of a thrown exception. So, if you choose to perform sophisticated exception handling within your JNI native code, instead of passing the exceptions back to Java, your code will be a lot less portable to the RNI.

The RNI defines the function `exceptionOccurred()`, which allows you to establish whether or not an exception is pending. This function simply returns a Boolean value, rather than an exception object. Thus, with RNI, you can only establish that an exception has been thrown, but not the type of the exception.

Sometimes, when you detect an exception, you can simply ignore it. With the JNI, you can clear all the pending exceptions with `ExceptionClear()` with the RNI, the function is `exceptionClear()`. These functions are generally useful in cases where an exception might be thrown to signify that an error has occurred, but the error has no real impact on the correct functioning of the program. For example, a native method might generate various exceptions to aid in debugging. You might not want these exceptions to propagate to either your native exception handler or back to your Java code. In this instance, explicitly clearing the pending exception allows you to continue debugging without exception handlers kicking in all over the place.

The third operation that you might want to perform when you detect an exception in native code is to print a stack trace. This is not commonly done, as it doesn't afford particularly flexible debugging, but you might find it useful for quick error diagnosis. With the JNI, you can call the `ExceptionDescribe()` function; it performs exactly the same function as calling `printStackTrace()` on an `Exception` object in Java. The corresponding RNI function is `exceptionDescribe()`. The stack trace is written to the system error-reporting stream, which might be `stderr` or `System.err` if you are using a standalone JVM or the Java Console if you are using Netscape Navigator. As with `printStackTrace()`, these functions are not particularly flexible, but they do report exactly where your program has been prior to throwing the exception. I find them quite invaluable for fast debugging at the outset of development projects.

Chapter 11. Threading

One of the major benefits to programming in Java is its inherent multi-threading capabilities. This enables you to write extremely powerful software capable of handling several tasks simultaneously. There are, of course, pitfalls with threading. One of the main ones, which this chapter discusses, revolves around the problem known as the *race condition*.

Race conditions exist when two threads are competing for a single resource. For example, if we had a GUI that performed drawing from different native methods, a race condition would inherently exist in the actual drawing loop as only one thread may access that at any given time. The section of code causes the problem is known as a *critical section* and should be somehow flagged as being susceptible to race conditions.

Synchronization and Monitors

Java provides the `synchronized` keyword that allows you to use a Java object as a semaphore, or in Java parlance, a *monitor*, which in turn defines a critical section of code and protects it from race conditions. For example, if we had a native method which only one thread can simultaneously enter, we can wrap the call with a `synchronized` block, or declare the method as being `synchronized`.

```
/** The native method - only one thread can access at a time */
public synchronized native void someNativeMethod();

/** Another native method -
control access to this via a synchronized block */
public native void someOtherNativeMethod();

/** Declare the monitor object to lock on */
Object lockObject;

/** Call the native method in a thread-safe manner */
synchronized( lockObject ) {
    someOtherNativeMethod();
  }
```

This technique ensures that the method cannot be invoked until the monitor lock is entered. This gives us a level of robustness, but in the case of intensive native methods with potentially long running time, it might reduce the effectiveness of your application operating in a multi-processing environment.

Within pure Java code, it is possible for us to wrap any chunk of Java code with a `synchronized`, even down to a single line of code. Can we do the same thing from native code instead of having to lock the entire native method?

Both the JNI and RNI make functions available for entering and exiting monitor locks. Under the JNI these are `MonitorEnter()` and `MonitorExit()`. Under the RNI, `monitorEnter()` and `monitorExit()`. For example, with the JNI:

```
JNIEXPORT void JNIEXPORT
Java_someClass_someNativeMethod( JNIEnv *env, jobject arg, jobject moni-
tor ) {
    /** Perform some code that doesn’t require protection */
    ...

    /** Protect the critical section */
    if ( env->MonitorEnter( monitor ) != 0 ) {
        fprintf( stderr, "Monitor enter failed! Potentially unstable re-
sults!\n" );
      }

    /** Execute the unsafe code */
    ...

    /** Leave the monitor */
    if ( env->MonitorExit( monitor ) != 0 ) {
        fprintf( stderr, "Monitor exit failed! The applica-
tion may hang!\n" );
      }

    /** Finish processing and exit... */
  }
```

The implementation of this method under the RNI is very similar.

```
RNIEXPORT void RNIEXPORT
someClass_someNativeMethod( HsomeClass *arg, HObject *monitor ) {
    /** Perform some code that doesn’t require protection */
    ...

    /** Protect the critical section */
    monitorEnter( (unsigned int)monitor );

    /** Execute the unsafe code */
    ...
    /** Leave the monitor */
    monitorExit( (unsigned int)monitor );

    /** Finish processing and exit... */
  }
```

In both cases, we have passed the Java object to lock on into the native method, but this object could be created within native code, perhaps as a global reference.

The RNI also features three other methods revolving around the topic of threading control: `monitorNotify()`, `monitorNotifyAll()` and `monitorWait()`. These methods simulate the methods declared in `java.lang.Object` for managing threads.

For much more in-depth information on multi-threading issues within Java, you should consult Doug Lea’s "Concurrent Programming in Java" (Addison Wesley, 1997).

Chapter 12. Memory Management

The Java Virtual Machine is a complex piece of software that performs a lot of housekeeping tasks that programmers usually have to do themselves. One key such area is memory management. Classic programming languages, such as C, allow the programmer to allocate and deallocate memory explicitly. The onus is therefore on you to handle this correctly, and, in particular, to free allocated memory when you have no further need of it. Forgetting to release allocated memory is a common problem in C programs; it results in programs that have memory leaks. And when a program asks for chunks of memory repeatedly without freeing the chunks appropriately, the accumulated memory leaks can cause the amount of free memory on the system to drop dramatically.

The JVM takes a different approach and manages your program's memory for you. That is, if you create a bunch of Java objects, the JVM cleans them up once you have no further need of them. This operation is known as *garbage collection* and is one of the reasons that Java is such an easy language to program with: someone else takes care of your memory management for you. Although garbage collection is generally regarded as a good thing, when you are writing native methods, it can sometimes be a positive pain in the neck.

There are two main problems that can arise with native methods and garbage collection. One occurs when the garbage collector moves or cleans up data that you are still referencing. Paradoxically, the other problem is that the garbage collector is disabled during the execution of native methods. We'll look separately at both of these issues and solutions to them in this chapter.

Persistent Objects

If you refer to a particular object within native code on a regular basis, it might make sense to try to cache the reference to this object, to avoid the time and effort involved in retrieving it repeatedly. What you need is a way to make the object persistent for a certain length of time. Fortunately, both the JNI and the RNI provide mechanisms for just this kind of functionality. The JNI distinguishes between global and weak global references, while the RNI calls them strong and weak references.

As you know, Java doesn't provide an explicit pointer to an object, but rather a reference to an object. A reference actually contains a pointer to the underlying location of an object within the JVM, but as a Java application programmer, you don't have or need access to this pointer. During garbage collection, the JVM may move the object to a more optimal location in memory, but the reference to the object does not change. Thus, from the application's point of view, the object is exactly where it always was, even though in reality it is not.

It's only when you start sharing references between Java code and native code that problems can arise. If you store a reference to an object in native code and then try to refer to it later, after the garbage collector has run, you may find that the garbage collector has relocated the object. In this case, you'll find that the reference is no longer valid, often with dramatic consequences.

In order to cache a reference to an object in your native code, you need to be able to tell the JVM not to move an object at all, or, in other words, to ignore the object during garbage collection. The idea is that you are *pinning* an object to a specific location in memory. This guarantees that the object will not be moved, so that you can rely on the cached references to it.

Both the JNI and the RNI support the following types of references:

Local references (JNI and RNI)

The default mode of operation for references within a native method. A local reference is valid for the lifetime of the currently executing native method. When the method exits, the local reference is freed automatically by the garbage collector. While the native method is executing, however, the garbage collector does not relocate or free any local references.

In other words, any objects that you are referencing within a native method are protected against the machinations of the garbage collector. This ensures that a reference that is valid in one line of code does not suddenly become invalid in the next line.

Global (JNI) or strong (RNI) references

Allow you to pin a reference for longer than the duration of a single native method invocation. When you make an object reference into a global or strong reference, you can rely on that reference not to be garbage collected until you explicitly unpin the reference. When you unpin a global or strong reference, it becomes a local reference again, so the garbage collector is free to relocate or remove it.

Weak global (JNI) or weak (RNI) references

Acts in the same way as a global or strong reference, but allows the garbage collector to move or free the underlying object. If the garbage collector moves the object, the reference tracks the location of the object, so that you can still use the reference. If the garbage collector frees the object, however, the reference is made to point to `NULL`. Note that weak global references are new in the JNI as of Java 2 Version 1.2.

Thus, to create a safe global variable that is not subject to the whims of the garbage collector, you simply need to turn a local reference into a global or strong reference, thereby pinning the reference. With the JNI, you call the `NewGlobalRef()` function, which takes a a `jobject` as an argument and returns a `jobject` value that is a global reference to the original object. The corresponding RNI fucntion is `GCNewHandle()`.

Because pinning an object causes the JVM to essentially ignore it, the JVM *never* garbage collects that object until you unpin it. If you forget to unpin a global or strong reference after you have finished with it, the resources used by that object are never released. If you continually pin objects and forget to release them, the performance of the JVM will degrade and it will consume more of your machine's resource. This a similar problem to memory leaks, but it degrades the ability of the JVM to function correctly much much faster.

To unpin a global or strong reference, you simply turn it back into a local reference. Then, when the native method exits, the garbage collector sees that the reference is now unpinned and thus available for garbage collection. To unpin a global or strong reference with the JNI, call `DeleteGlobalRef()`. The corresponding RNI function is `GCFreeHandle()`.

If you don't need the absolute safety of a global or strong reference, and you only need to know whether a reference has been relocated, consider using a weak global or weak reference instead. Such a reference does not protect data from being moved by the garbage collector, but the reference does track the desired object. In other words, if the garbage collector moves an object, the weak (global) reference is updated to point at its new location in memory. This is extremely useful for making Java data persistent across native methods.

As of Java 2 Version 1.2, you can create weak global reference with the JNI by calling the `NewWeakGlobalRef()` function. If successful, the function returns a new `jobject` that is a weak global reference to the original object. If the referent object is moved by the garbage collector, the weak global reference is updated to point at its new location. However, if the referent object is freed by the garbage collector, the weak global reference is set to `NULL`, allowing you to detect that is has been removed. Thus, when you are using a weak global reference, you always need to check that it isn't `NULL` before you use it. The RNI defines the `GCGetPtr()` function for creating weak references, which function in exactly the same manner. When you want to release a weak global or weak reference, you simply call `DeleteWeakGlobalRef()` with the JNI or `GCFreePtr()` with the RNI.

Before you go off and start creating all manner of persistent references, let me warn you about a potential problem. Using global variables makes it much more difficult to write multi-thread safe code, since multiple running threads can simultaneously alter the value of the object referenced by the variable. As a consequence, you would be wise to use persistent references sparingly, if at all. If you do use them, you need to take extra caution and design your native methods with multithreading in mind. Wrapping your native method calls in `synchronized` blocks can certainly help avoid deadlock or race conditions, but this alone does not allow you necessarily to guarantee the validity of an object referenced by a global variable.

Controlling the Garbage Collector

When your Java code enters a native method, the JVM suspends garbage collection. The JVM does this because garbage collection during native code execution can be difficult to perform safely and at acceptable speeds. In most cases, native methods return very quickly, so the fact that garbage collection has halted is not necessarily a problem since it is automatically reenabled upon leaving the native method. However, if you have a native method that does some heavy-duty processing or runs for a long period of time, the disabling of the garbage collector can be a drain on memory, as redundant objects are never freed up. To alleviate this problem, the JNI and the RNI both allow you to explicitly control various aspects of memory management during native method execution.

JNI Memory Management

With the JNI, you can explicitly remove a local reference to an object by calling `DeleteLocalRef()`. Obviously, once you have called `DeleteLocalRef()` on a reference, you cannot rely on that object being available anymore. It may not be removed immediately, but don't plan on reusing it unless you want to see strange errors in your program.

To consider how this function might be useful, say you are performing lots of string operations within native code, for example, running a lexical analyzer. Your program might only need a small number of strings to operate correctly, with the vast majority of strings needed only temporarily during analysis. However, these temporary string are not garbage collected until the native method exits, since a local reference exists for each unwanted string. In this case, calling `DeleteLocalRef()` on each unwanted string removes the local reference almost immediately, during the native method's execution, rather than when the native method exits. You can think of this function as a JNI equivalent to `free()` in C.

As of Java 2 Version 1.2, the JNI provides additional functionality that allows more fine-grained control over the allocation and operation of local references. It is actually possible to create too many local references in a native method, so that further local reference creation fails. For example, say you are manipulating a large array. If you are allocating a local reference for each array element as you iterate through the array, but not releasing those local references explicitly with `DeleteLocalRef()`, you may exhaust the supply of local references. The new `EnsureLocalCapacity()` function allows you to ensure that at least a certain number of local references can be created within the current native method. The default number of local references that the JVM allocates is 16.

When a program enters a native method, the JVM creates a new local reference frame, which basically sets the scope of local references to be within that particular frame. When the native method is exited, the local reference frame is deleted, freeing up all the local references created within it. As of Java 2 Version 1.2, the `PushLocalFrame()` and `PopLocalFrame()` functions allow you to explicitly create and destroy a new local reference frame within which local references are allocated. The new frame is created with a given number of local references that can be allocated. For example, if you are scanning through a multi-dimensional array of objects that you want to create local references for, performance may suffer if you implement your code using a single local reference frame as shown in the following code fragment:

```
/* Default local reference frame is active upon native method entry */
...
```

```
/* Scan the array */
for ( int i = 0 ; i < arrayWidth ; i++ ) {
    for ( int j = 0 ; j < arrayHeight ; j++ ) {
        /* A new local reference is allocated for the array ele-
ment at [i,j] */
      }
  }
```

As the number of local references mounts up through scanning the array, performance issues may arise as the local reference frame holds more and more references. A solution around this problem is to create and destroy a local reference frame for each “slice” of the array. For example:

```
/* Default local reference frame is active upon native method entry */
...

/* Scan the array */
for ( int i = 0 ; i < arrayWidth ; i++ ) {
    /*
     * Allocate a new local reference frame for the required number of
     * references
     */
    env->PushLocalFrame( arrayHeight );

    /* Scan a slice of the array */
    for ( int j = 0 ; j < arrayHeight ; j++ ) {
        /*
         * A new local reference is allocated for the array
         * element at [i,j]
         */
      }

    /* Destroy the current local reference frame and free those references */
    env->PopLocalFrame( NULL );
  }
```

One final operation you can perform with the JNI as of the Java 2 Version 1.2 is copying a reference, using the `NewLocalRef()` function. This allows you to create a new local reference that is a copy of an exising local or global reference.

With the collection of JNI functions we’ve discussed here, it is possible to achieve a certain level of fine-grained memory management within Java’s existing garbage collection framework. With careful application, these features can be used to provide leaner Java programs, in terms of their memory requirements and usage.

RNI Garbage Collection Control

Memory management in RNI native methods is a bit different than with JNI methods, as the RNI provides a way to enable the garbage collector during native method execution. Prior to entering a computationally- or memory-intensive section of code in an RNI native method, you can call `GCEnable()` to enable the garbage collector, causing it to run simultaneously with your native code. And when that section of code is done, or prior to returning from the native method, you must then call `GCDisable()` to disable the garbage collector.

It is extremely important to ensure that every call to `GCEnable()` has a matching call to `GCDisable()` because returning from a native method call causes `GCEnable()` to be called automatically. If you already have the garbage collector enabled (i.e., you haven't called `GCDisable()`, you may experience bizarre problems in the operation of your code or outright crashes of the JVM.

The ability to enable garbage collection explicitly within a native method brings up the problem of the interaction between native code and the garbage collector again. For example, say you have a pointer to an object that you are using within your native code. As we've already discussed, during garbage collection this object may be moved to another location in memory, making your pointer invalid. While this is a problem, you really do need garbage collection to occur. Fortunately, the RNI defines a way in which you can have your cake and eat it too.

With the RNI, you can declare a structure known as a `GCFrame` and push objects into it that the garbage collector has to leave alone. For example, you can declare a C `struct` that contains the pointers to all the objects that you do not wish to be moved. You then simply associate this data structure with the `GCFrame` using `GCFramePush()`.

Pushing this information onto the garbage collector stack actually does allow the garbage collector to move the objects, but the pointers contained within the `struct` are automatically updated with the new locations of the objects within memory. Therefore, instead of referring to the original pointer when you wish to manipulate objects, you should use the pointers stored within the `struct`.

Finally, once you have disabled garbage collection, you must deallocate the `GCFrame` you have been using, to allow the garbage collector to function as normal when you exit from the native method. To perform this operation, you should call `GCFramePop()`. As with the `GCEnable()`/`GCDisable()` pair, you must remember to pop every `GCFrame` that you push! Here is a code fragment that shows the use of a `GCFrame`:

```
RNIEXPORT void RNICALL
SymbolStone_someMethod( struct HSymbolStone *stone ) {

    /*
     * Declare a structure containing the pointer(s) that the
     * garbage collector has to leave alone.
     */
    struct {
        HSymbolStone *safeStone;
      } gcProtect;

    /* Declare the GCFrame */
    GCFrame gcf;

    /* Push the frame and pointer structure into the garbage collector */
    GCFramePush( &gcf, &gcProtect, sizeof( gcProtect ) );

    /* Store the object pointer(s) within the protected structure */
    gcProtect.safeStone = stone;

    /* Enable garbage collection */
    GCEnable();

    /*
     * At this point, the object pointed at by the stone parameter
     * might be moved. The following statement has a good chance of
     * crashing the VM, so it is commented out:
     *
     * SymbolStone_t *sptr = (SymbolStone_t *)stone->_pData;
     */

    /*
     * Instead, you should refer to the stone pointer stored within
     * the protected structure. This pointer tracks any changes
     * made to the object’s location by the garbage collector.
     */
    SymbolStone_t *safesptr = (SymbolStone_t *)gcProtect.safeStone->_pData;

    /* Disable garbage collection... */
    GCDisable();

    /* ...and free up the pushed GCFrame */
    GCFramePop( &gcf );

    /* And now we can safely exit from this native method! */
  }
```

As this example illustrates, computationally-intensive RNI native methods can be made efficient in terms of memory consumption, with some careful programming.

Chapter 13. Embedding a JVM

To this point, our discussion of native method programming has been about using native code to implement certain Java methods in C or C++. There is, however, another way to use native code and the JNI or RNI: you can embed a JVM within a C/C++ program and manipulate Java classes and objects from within that program. This mode of interaction between native code and Java is more far-reaching in its application and power than simple native method development. For example, a Java-enabled web browser uses this form of interaction to allow you to run applets within web pages. In this case, the natively written web browser has a JVM embedded within it and drives the operation of the JVM from native code.

Both the JNI and the RNI allow you to embed a JVM within a program, but the way in which they do this differs considerably. The JNI uses a subsidiary API known as the Invocation API that is distinct from the core JNI specification itself. The RNI, however, simply provides two functions that hook you directly into the JVM.

Using the JNI Invocation API

With the JNI Invocation API, the core operations for embedding a JVM in a program are creating a new JVM, destroying a JVM, attaching a thread to a running JVM, and detaching from a JVM. Once you have created a JVM and, if necessary, attached a thread, you simply perform JNI operations to execute any Java code you want to run.

Creating a JVM

The creation of a JVM is a fairly simple task which is performed by using the Invocation API method called `JNI_CreateJavaVM()`. Prior to invoking this method, you will need to perform some variable declarations and setup of the JVM environment in order for the JVM to function in a useful way.

Configuring the JVM

Prior to creating a JVM for use within your native application, some configuration is required before the JVM will function in a useful way. A short piece of stub code illustrating the variables required in JVM creation and configuration is shown below.

```
#include <jni.h>      /** The Invocation API methods are declared here */

/** A pointer to an object encapsulating a Java VM representation */
JavaVM *jvm;
```

```
/** A pointer to a valid JNI environment */
JNIEnv *env;

/** A structure which contains the actual settings the JVM will operate un-
der */
JDK1_1InitArgs vmArgs;

...
```

Of these declarations, the `JDK1_1InitArgs` structure is the particularly interesting one. This structure contains a number of fields which can be set changing the way in which the JVM will operate after creation. The following table shows the contents of the `JDK1_1InitArgs` structure.

```
typedef struct JDK1_1InitArgs {
    jint version;

    char **properties;
    jint checkSource;
    jint nativeStackSize;
    jint javaStackSize;
    jint minHeapSize;
    jint maxHeapSize;
    jint verifyMode;
    char *classpath;

    jint (JNICALL *vfprintf)(FILE *fp, const char *format, va_list args);
    void (JNICALL *exit)(jint code);
    void (JNICALL *abort)();

    jint enableClassGC;
    jint enableVerboseGC;
    jint disableAsyncGC;
    jint verbose;
    jboolean debugging;
    jint debugPort;
} JDK1_1InitArgs;
```

The majority of these fields may be familiar to you if you have used the `java` program shipped with the Sun JDK or JRE. This program allows you to set various JVM operating parameters from command line arguments. By sheer coincidence, many of those parameters correspond to the fields within this structure!

It is important that all the fields within this structure are initialized to valid values prior to creating a JVM otherwise it is likely that either initialization will silently fail or the application will just keel over.

Fortunately, there's a handy method defined in the Invocation API that will populate the structure with platform-specific values. This method is called `JNI_GetDefaultJavaVMInitArgs()` and it's use can be demonstrated with the following short program which populates the structure then displays the default values.

```
/**
 * showArgs1.cpp - Populates a JDK1_1InitArgs structure with default values
 */

#include <stdlib.h>
#include <jni.h>

int main() {

    JDK1_1InitArgs vmArgs;

    /** Get the default VM arguments */
    JNI_GetDefaultJavaVMInitArgs( &vmArgs );

    /** Print them all out */
    printf( "JNI Information\n" );
    printf( "===============\n" );
    printf( "\tJDK version: %#010x\n", vmArgs.version );
    printf( "\n" );

    printf( "Properties and Classpath\n" );
    printf( "========================\n" );
    printf( "\tProperties: 0x%08x\n", vmArgs.properties );
    printf( "\tClasspath: %s\n",
             ( vmArgs.classpath == NULL ) ? "not set" : vmArgs.classpath );
    printf( "\n" );

    printf( "Class Loader Information\n" );
    printf( "========================\n" );
    printf( "\tCheck Source? %s\n",
             ( vmArgs.checkSource == JNI_TRUE ) ? "yes" : "no" );
    printf( "\tVerify Classes? %s\n",
             ( vmArgs.verifyMode == JNI_TRUE ) ? "yes" : "no" );
    printf( "\n" );

    printf( "Garbage Collector Information\n" );
    printf( "=============================\n" );
    printf( "\tEnable Class GC? %s\n",
             ( vmArgs.enableClassGC == JNI_TRUE ) ? "yes" : "no" );
    printf( "\tEnable Verbose GC? %s\n",
             ( vmArgs.enableVerboseGC == JNI_TRUE ) ? "yes" : "no" );
    printf( "\tDisable Async GC? %s\n",
             ( vmArgs.disableAsyncGC == JNI_TRUE ) ? "yes" : "no" );
    printf( "\n" );

    printf( "Debugging Information\n" );
    printf( "=====================\n" );
    printf( "\tRemote Debugging Enabled? %s\n",
             ( vmArgs.debugging == JNI_TRUE ) ? "yes" : "no" );
    printf( "\tRemote Debugging Port: %d\n", vmArgs.debugPort );
    printf( "\n" );

    printf( "Memory Allocation Information\n" );
    printf( "=============================\n" );
    printf( "\tNative Thread Stack Size ( bytes ): %d\n",
             vmArgs.nativeStackSize );
    printf( "\tJava Thread Stack Size ( bytes ): %d\n",
             vmArgs.javaStackSize );
    printf( "\tMinimum JVM Heap Size ( bytes ): %d\n",
             vmArgs.minHeapSize );
    printf( "\tMaximum JVM Heap Size ( bytes ): %d\n",
             vmArgs.maxHeapSize );
    printf( "\n" );

    printf( "Function Hooks\n" );
    printf( "==============\n" );
    printf( "\tvfprintf() @ %x\n", vmArgs.vfprintf );
    printf( "\texit() @ %x\n", vmArgs.exit );
    printf( "\tabort() @ %x\n", vmArgs.abort );

    exit( 0 );
  }
```

After compiling and running this program on Linux with JDK-1.1.7, the output of the program shows the following default values.

```
JNI Information
===============
JDK version: 0x00010001

Properties and Classpath
========================
Properties: 0x00000000
```

```
Classpath: /opt/jdk1.1.7/lib/i386/green_threads/../../../classes:/opt/jdk1.1.7/lib/i386/gree

Class Loader Information
========================
Check Source? no
Verify Classes? yes

Garbage Collector Information
=============================
Enable Class GC? yes
Enable Verbose GC? no
Disable Async GC? no

Debugging Information
=====================
Remote Debugging Enabled? no
Remote Debugging Port: 0

Memory Allocation Information
=============================
Native Thread Stack Size ( bytes ): 131072
Java Thread Stack Size ( bytes ): 409600
Minimum JVM Heap Size ( bytes ): 1048576
Maximum JVM Heap Size ( bytes ): 16777216

Function Hooks
==============
vfprintf() @ 0
exit() @ 0
abort() @ 0
```

It is extremely important to note the default states of these variables as altering them to setting other than the default can cause unexpected stability or performance problems in your applications. As I elaborate on each of the variables in the `JDK1_1InitArgs` structure, I shall point out what some of these caveats are.

The `version` field contains a hexadecimal representation of the current JNI version. For example, a JDK-1.1 JVM will return `0x00010001`. This value corresponds to what the JNI method `GetVersion()` returns. You can decode this value by taking the high 16 bits of the value to represent the *major* version of Java and the lower 16 bits to represent the *minor* Java version. You can assign new values to this variable causing the JVM to change modes of operation. For example, there have been several enhancements added to the JNI and Invocation API for Java-2. By assigning an appropriate value to the `version` variable, you can “turn on” and “switch off” these values depending on which version of the

API your code conforms to. Setting this value after JVM creation is not recommended. At best it will have no effect, at worst it will crash your application.

You can also assign values to a group of variables that dictate the quantity of memory the JVM will allocate for its own internal usage, *i.e.*, its *stack* and *heap* sizes. These variables are `nativeStackSize`, `javaStackSize`, `minHeapSize` and `maxHeapSize` and should be assigned an integerial value specified in *bytes*. Alteration of these values can be useful if you know in advance that your application has a high turnover of objects, for example. By allocating more heap space to the JVM, your application may run faster by requiring less dynamic memory allocation for objects. However, it is also possible to allocate too much space for the JVM which will cause your machine to page and swap more often causing an degradation in overall system performance.

Another group of variables defined within this structure regulate operations on bytecode when loading classes. These variables are `checkSource` and `verifyMode` and signify whether or not the source of a Java class is newer than the class when the class is loaded and whether or not the bytecode being loaded should be passed through the *bytecode verifier* when loaded[1]. Both variables accept either the values of `JNI_TRUE` or `JNI_FALSE` specifying their mode of operation.

You can also exert some level of control over the global garbage collector settings by altering the variables named `enableClassGC`, `enableVerboseGC` and `disableAsyncGC`. All three variables take a value of either `JNI_TRUE` or `JNI_FALSE` to specify the new mode of operation. Each variable has a slightly different effect on JVM garbage collecting in that `enableClassGC` allows garbage collection of unused class information, `enableVerboseGC` enables the displaying of what the garbage collector is doing as it does it and `disableAsyncGC` alters the way in which the garbage collector runs. Garbage collection usually occurs within a separate low-priority background thread and runs asynchronously of the main application. The main effect of this is that unused classes and objects can be removed quietly from the JVM without any obvious performance impacts on your applications. By disabling asynchronous garbage collection, the garbage collection operations may run slightly faster, but your application may be suspended during garbage collection. If a large quantity of redundant objects and classes are being garbage collected, this may render your application almost unusable.

It is possible to configure the remote debugging capabilities of the JVM through two variables defined within this structure. The `debugging` variable may be set to either `JNI_TRUE` or `JNI_FALSE` indicating whether or not remote debugging capabilities are to be enabled or disabled. You may also specify the port on which a Java debugger is configured to be listening on by assigning the port number, in integerial form, to the `debugPort` variable. Another useful variable within the `JDK1_1InitArgs` structure is `verbose` which simply verbosely prints out what the JVM is up to. This is useful to watch class files loading up to ensure the application is finding the correct classes to use. This variable takes either `JNI_TRUE` or `JNI_FALSE` as its new value enabling or disabling verbose operation respectively.

The `JDK1_1InitArgs` structure also defines three *function hooks* which allows you to redefine the operation of three JVM operations. The function hooks allow you to redefine what happens when the JVM aborts, it prints out some diagnostic information or it exits. For example, if you wished to redirect the output of the JVM’s operation as it runs verbosely, you can implement a customized `vfprintf()` function which writes the information out to a file. The following code stub illustrates this principle.

```
#include <jni.h>
#include <stdio.h>

/** The JVM initialization arguments */
JDK1_1InitArgs vmArgs;

static FILE *debugFile = NULL;
static jboolean fpCanWrite = JNI_TRUE;

/** Customized vfprintf() function that writes any diagnos-
tic data to a file */
jint (JNICALL myvfprintf)( FILE *fp, const char *fmt, va_list args ) {

    jint rv;

    /** If the file isn’t already open..... */
    if ( debugFile == NULL ) {
        /** Have we tried opening it before? */
        if ( fpCanWrite == JNI_TRUE ) {
            /** No, so open the file up... */
            if ( ( debugFile = fopen( "vmdebug.out", "w" ) ) == NULL ) {
                /** Yikes! We can’t open the file! Flag it as unopenable */
                fpCanWrite = JNI_FALSE;
                return -1;
              }
          }
      }

    /** Pass the data to the real vfprintf() */
    rv = vfprintf( debugFile, fmt, args );

    /** Flush the file and return the value from the “real” vfprintf() */
    fflush( fp );
    return rv;
  }

/** The main application body */
int main( int argc, char **argv ) {

    /** Get the default VM arguments */
    JNI_GetDefaultJavaVMInitArgs( &vmArgs );
    /** Turn on verbose operation of the JVM */
    vmArgs.verbose = JNI_TRUE;

    /** Redirect vfprintf() to our customized function */
    vmArgs.vfprintf = myvfprintf;

    ...
  }
```

This example debugging function uses a static filehandle to write the data out to a file called `vmdebug.out`. You could write the function to open and close a local filehandle for each debug message printed out, but this may prove detrimental to the overall performance of your application. The other two function hooks can be used in exactly the same way as the example above, obviously after changing the function prototypes and return type to suit.

The only remaining variables in this structure that I have not discussed pertain to the setting of system properties and the setting of the path in which class files may be found. These two variables are named `properties` and `classpath` respectively.

The `classpath` variable is by far the most important variable in the entire structure as it specifies a list of directories, zip and `jar` files that class files can be found in. It is important to note that this includes all the system classes such as those contained within the `java.lang` package.

In JVMs conforming to JDK-1.1.3 or earlier, the default value of the `classpath` variable was `NULL` as shown in the default values output above. This had the unfortunate effect that if you did not explicitly set the `classpath` variable in your application prior to calling `JNI_CreateJavaVM()`, none of the system classes could be found and your application would die immediately. In JVMs conforming to JDK-1.1.4 and later, the `classpath` field defaults to the value of the `CLASSPATH` environment variable as set in your current shell which is slightly more sensible. The major problem inherent in the use of the `CLASSPATH` environment variable is that if you ship your application to another user, you cannot guarantee that their environment is setup in the same way as yours which may lead to your application failing.

Another snag with setting the `classpath` value within the `JDK1_1InitArgs` structure is that the actual format of the classpath varies depending on the platform upon which the application is being run. For example, the directory separator for UNIX is a forward slash (`/`), but on Windows a backslash is used which needs to be escaped (`\\`). Similarly, on UNIX the separator between class file locations is a colon (`:`) but Windows uses a semi-colon (`;`).

My usual solution to this problem is to use preprocessor definitions and specify both forms. This ensures that your application will continue to run provided you keep the variables in synchronization! For example

```
/** Virtual Machine initialization arguments */
JDK1_1InitArgs vmArgs;

/** Setup the classpath depending on platform */
#ifdef WIN32        /** Windows-only */
vmArgs.classpath = "c:\\jdk1.1.7\\lib\\classes.zip;c:\\dev\\classes";
#else /** WIN32 */
#ifdef UNIX
vmArgs.classpath = "/opt/jdk1.1.7/lib/classes.zip:/opt/descarte/dev/classes";
#else /** UNIX */
#ifdef MACOS
vmArgs.classpath = "Sys-
tem Folder: Java: Classes;Disk: Descarte: Dev: Classes";
#endif /** MACOS */
#endif /** UNIX */
#endif /** WIN32 */
```

A similar work-around solution for JVMs of version JDK-1.1.3 or earlier would be to just pick up the `CLASSPATH` environment variable. For example

```
vmArgs.classpath = getenv( "CLASSPATH" );
```

Since the `classpath` variable is simply a string, you can concatenate strings together using `strcat()` or even `sprintf()`. This allows you to build classpath values suitable for your environment and hopefully helps you avoid hard-coded values.

The final variable within the `JDK1_1InitArgs` structure is that of `properties`. This variable contains an array of name & value pairs specifying the values of given properties. For example, if you were using a mail reader which allowed you to set properties including one containing your email address, the name / value pair would be

```
email.emailAddress=descarte@arcana.co.uk
```

Within the context of the Invocation API, properties of this type can be set by creating an array of strings containing these pairs. To create a property list with the above property, you would write

```
/** The JVM initialization structure */
JDK1_1InitArgs vmArgs;

/** Setup the properties that you want to pass into the JVM */
char *properties[] = {
    "email.emailAddress=descarte@arcana.co.uk",
    NULL
  };

/** Assign your properties to the structure for initialization */
vmArgs.properties = properties;
```

You *must* make sure that you have `NULL`-terminated the list of properties. Similarly, as with the setting of the `classpath` variable, any properties that refer to files must conform to the format expected for that particular platform. Again, preprocessor directives could be used to control the correct specification for multiple platform support.

The Java-2 Invocation API takes a slightly different approach to the allocation and specification of arguments to be used when initializing a JVM. The data structure of type `JDK1_1InitArgs` is quite JVM-specific and results in a certain amount of unportability and general inflexibility regarding the specification of a JVM configuration. The Java-2 defines a new opaque format for configuration specification revolving around the structure of type `JavaVMInitArgs`. This structure is defined as

```
typedef struct JavaVMInitArgs {
    jint version;

    jint nOptions;
    JavaVMOption *options;
    jboolean ignoreUnrecognized;
} JavaVMInitArgs;
```

in the Java-2 version of `jni.h`. The way in which this structure is used to to allocate a single instance of `JavaVMInitArgs` then further allocate the required number of `JavaVMOption` structures to contain information on each particular configuration option.

The `JavaVMOption` structure is defined in the following way

```
typedef struct JavaVMOption {
    char *optionString;
    void *extraInfo;
} JavaVMOption;
```

and is essentially a strictly type name / value pair in that the name of the option is assigned to the `name` variable and the appropriate value of that configuration option should be set in the `value` union.

Using this system is relatively simple and if you wished to configure a verbosely executing JVM with a given classpath, you could write it as

```
/**
```

```
 * showArgs2.cpp - Populates a VMInitArgs structure with default values
 */

#include <stdlib.h>
#include <jni.h>

int main() {

    /** Arguments used to configure the JVM */
    JavaVMInitArgs vmArgs;

    /** Array of options for the JVM configuration. We need two elements */
    JavaVMOption vmOptions[2];

    /** Setup the base configuration */
    vmArgs.version = 0x00010002;
    vmArgs.nOptions = 2;

    /** Setup the classpath option */
    char *classpath = getenv( "CLASSPATH" );
    sprintf( classpath, "-classpath %s", classpath );
    vmOptions[0].optionString = classpath;

    /** Setup the verbose option... */
    vmOptions[1].optionString = "-verbose:gc,class";

    /** Set the options within the base configuration */
    vmArgs.options = vmOptions;

  }
```

The major caveats with this form of configuration are that you must correctly specify the names of each configuration option for it to work correctly. A further problem is that there is no way of enumerating the options available for each JVM. Options that are not portable between JVMs are prefixed with an underscore (_). For example, the initial and maximum heap size values are now called `_ms` and `_mx`.

However, all JVM implementations must support a small, but important, subset of configuration options as standard. These are `classpath`, `properties`, `verbose`, `vprintf`, `exit` and `abort`. All these options conform to the descriptions given previously with the exception of `verbose`. This parameter now takes a comma-separated string containing the aspects of the JVM that you wish to report verbosely on rather than a simple on-off toggle. Some sample values for this string are `gc` for verbose garbage collection tracing, `jni` for verbose JNI tracing and `class` which reports class loading activities.

One useful aspect of the Java-2 additions to the `JavaVMInitArgs` structure is that by specifying a value of `JNI_FALSE` to the `ignoreUnrecognized` field, any badly specified arguments will simply be ignored.

A final bonus for using the Java-2 approach for JVM configuration is that the concept of `JNI_GetDefaultJavaVMArgs()` does not exist anymore and there is no point in calling that method.

Really Creating The JVM!

Finally! Now that we have explored the various options available to you when you wish to create a new Java Virtual Machine from within your native application, it's time to actually create it!

Unfortunately, after all the build up and excitement of configuring all the various options available to you, the actual JVM creation is extremely easy and requires you to invoke a single method, namely `JNI_CreateJavaVM()`. I shall detail a short example in a moment that illustrates the whole procedure from specifying some options within the `JDK1_1InitArgs` structure down to actually creating the JVM and testing that it works *via* this small Java class.

```
public class testClass {

    /** Static method which simply prints out a short message */
    public static void printMessage() {
        System.out.println( "Hello from the Java test class!" );
      }
  }
```

The corresponding native application which will use this Java class is

```
/**
 * invtest.cpp
 *
 * Test embedding a JVM into an application using the Invocation API in
 */

#include <stdlib.h>
#include <string.h>
#include <jni.h>

/** The Java Virtual Machine representation */
JavaVM *jvm;

/** A JNI Environment pointer */
JNIEnv *env;
```

```
/** The initialization arguments for the JVM creation phase */
JDK1_1InitArgs vmArgs;

/** The main application */
int main() {

    /** Set the basic classpath to point at the system class files */
    char *classpath = "e:\\jdk1.3\\jre\\lib\\rt.jar";

    /** Concatenate the custom directories to the classpath */
#ifdef WIN32
    strcat( classpath, ";..\\..\\..\\classes" );
#else
    strcat( classpath, ":../../../classes" );
#endif /** WIN32 */

    /** Fetch the default VM initialization arguments */
    JNI_GetDefaultJavaVMInitArgs( &vmArgs );

    /** Configure the JVM initialization arguments for this application */
    vmArgs.classpath = classpath;
    vmArgs.verbose = JNI_TRUE;
    vmArgs.enableVerboseGC = JNI_TRUE;

    /** Create a new JVM */
#ifdef JNI_VERSION_1_2
    JNI_CreateJavaVM( &jvm, (void **)&env, &vmArgs );
#else
    JNI_CreateJavaVM( &jvm, &env, &vmArgs );
#endif /** JNI_VERSION_1_2 */
    fprintf( stderr, "here\n" );


    /** Test that the JVM works by invoking a method in a test class */
    jclass testClassClassblock = env->FindClass( "testClass" );
    if ( testClassClassblock == NULL ) {
        env->FatalError( "testClassClassblock is NULL! Aborting!" );
      }

    jmethodID printMessageMethodID =
        env->GetStaticMethodID( testClassClassblock, "printMessage", "()V" );
    if ( printMessageMethodID == NULL ) {
        env->FatalError( "printMessageMethodID is NULL! Aborting!" );
      }
    env->CallStaticVoidMethod( testClassClassblock, printMessageMethodID );

    /** Destroy the JVM and deallocate all its resources */
    jvm->DestroyJavaVM();
  }
```

As you can no doubt see from the latter lines in this example, invoking the static method with the test class is using the standard JNI methods that we have already discussed within the earlier section of this chapter.

Attaching and Detaching Threads

The JVM creation technique that I outlined in the previous section suits single-thread applications perfectly. However, if you wish to integrate a JVM into a multi-threaded program or more directly into an operating system, you will rapidly hit problems.

The major difficulty with integration of a JVM into multi-threaded programs is the fact that the `JNIEnv` function table pointers are generally invalid across multiple threads. That is, each thread will use a completely different `JNIEnv`[2].

However, being able to use a JVM created within your application from separate threads does seem like a fairly useful thing to be able to do. For example, an application depending on constantly streamed data from a network might read this data in a low priority background thread using a native method. It would be extremely useful to be able to simply invoke a Java method to trigger some sort of update mechanism within the application when new data is received. However, if the JVM embedded within the application had not been created within the network reading thread, how does it invoke the appropriate Java method safely?

The answer to this problem lies in two methods called `AttachCurrentThread()` and `DetachCurrentThread()`. `AttachCurrentThread()` does pretty much what the name implies. It hooks the current thread into the JVM which you have already created and populates a `JNIEnv` data structure with data pertinent to that thread. At this point, you can use the new `JNIEnv` structure to issue JNI calls. For example,

```
/** Globally declared JVM embedded within the application */
extern JVM *jvm;

/**
 * Handles network reads and screen updates accordingly. This function is
 * the entry point for a native thread...
 */
void readAndRedraw() {

    /** Return code from the network */
    int returnCode = 0;

    /** Make connection to network */
    ...

    /**
     * The JNIEnv environment pointer for this thread. This will be populated
     * after thread attachment
     */
    JNIEnv *env;

    /**
     * The arguments for the JVM. This will also be populated after thread
     * attachment
     */
    JDK1_1AttachArgs *threadArgs;

    /** Attach this thread to the JVM before any JNI calls can be made! */
    jvm->AttachCurrentThread( &env, &threadArgs );

    /** Some commonly used Java information about the updater class */
    jclass updaterClassblock =
        env->FindClass( "updater" );
    jmethodID doUpdateMethodID =
        env->GetMethodID( updateClassblock, "doUpdate", "()V" );

    /** Some commonly used Java information about the main Java class */
    jclass realTimeAppClassblock = env->FindClass( "RealTimeApp" );
    jfieldID updaterFieldID =
        env->GetStaticFieldID( realTimeAppClassblock, "_updater",
                               "Lupdater;" );
    jobject _updater =
        env->GetStaticField( realTimeAppClassblock, updaterFieldID );

    /** Infinitely loop until we get a network disconnect */
    while ( returnCode != 666 ) {
        /** Read data from network */

        /** Perform different actions depending on the data coming in */
        switch ( returnCode ) {
            case 333: {
                /** Trigger the update */
                env->CallObjectMethod( _updater,
                                       doUpdateMethodID );

                break;
              }
            default: {
                break;
              }
          }
      }

    /** We’ve exited the loop at this point, so, detach the thread.... */
    jvm->DetachCurrentThread();
  }
```

When `AttachCurrentThread()` is called, two data structures are created, the afore-mentioned `JNIEnv` information and a structure of type `JDK1_1AttachArgs` which is populated with information. Under JDK-1.1, this structure is empty and contains no information whatsoever!

However, under the Java-2 JNI Specification `AttachCurrentThread()` takes a new third parameter conforming to the version of function table the second parameter is to be populated with. The third parameter will take values of either `0x00010001` which indicates a JDK-1.1 function table and `0x00010002` signifying a Java-2 function table. However, it still appears that nothing useful in placed in the resulting data structure even in Java-2!

Finally, it is extremely important to detach any threads you have attached to the JVM once you’re finished with them otherwise invocations of `DestroyJavaVM()` will block until the thread is detached. In the example above, the thread would die without detaching the thread from the JVM and this would cause problems with JVM operation and eventual destruction. It is possible to diagnose whether or not the current thread is attached to a particular JVM by using the Java-2 specific `IsAttached()` method. This will return `JNI_TRUE` if the current thread is attached to the given JVM and `JNI_FALSE` if not. Please remember that a thread can only be attached to *one* JVM at any time!

Destroying the JVM

The final stage of JVM interaction that should be taken from a native application is to explicitly destroy the JVM once your application has finished using it. This ensures that all resource is completely deallocated. Destruction of the JVM is very simple to achieve and is achieved by the Invocation API method called `DestroyJavaVM()`.

There are some catches to using this method in that the thread attempting to destroy the JVM must be the only thread still running in the JVM. That is, you should not have attached any new thread and not detached them otherwise the `DestroyJavaVM()` call will block until all other threads die. Also, JDK-1.1 virtual machines do not support the unloading, or destruction, of a JVM. This has the side-effect that the return status of `DestroyJavaVM()` on a JDK-1.1 JVM is always `-1` indicating failure.

Furthermore, the JDK-1.1 placed the restriction upon you that the thread in which the JVM was created must always destroy the JVM. This restriction has been lifted in Java-2 in which any thread can destroy the JVM. However, that thread must be the only one currently attached to the JVM at that point otherwise, the `DestroyJavaVM()` call will block as usual.

Compiling Invocation API Programs

The way in which Invocation API programs are compiled and executed differs somewhat from the usual native methods build style. By using the Invocation API, you are embedding Java into an existing program, not implementing certain Java methods using native code. Therefore, the focus of the build is moved away from creating an object, or shared library, that is “pulled into” the JVM at runtime to compiling a standalone program that simply links with the JVM as well as other libraries.

Therefore, if all that is required to compile our standalone code that uses the Invocation API is to link with some additional libraries, what are these and where are they? Under JDK-1.1.7, the library in question for the Sun JVMs is called `libjava.so` on UNIX platforms and `javai.lib` on Windows and OS/2. This library can be found in the `lib` directory in your Java installation[3]. Simply add that library to the list of libraries you are linking with to ensure that all the Invocation API functions are in place and that Java will be present within your programs.

For Java-2 platforms, things get a little more complicated. On UNIX, the libraries in question can involve `libjvm.so` and `libhpi.so`[4]. Under Win32, the library is simply `jvm.lib`. These libraries are typically found within the `jre` subdirectory of your Java installation.

To help clarify this, you should examine the sample `Makefile`s and Visual Studio project files provided with the example code.

Registering Native Methods Programmatically

A final topic of using the JVM in an embedded form that we shall cover is the topic of being able to statically link native methods into your application. This essentially dispenses with the need to call `System.loadLibrary()`, but requires you to *register* your native methods with the JVM programmatically. This technique is not specifically categorised as being part of the Invocation API, but really only useful in this context.

The idea of registering your native methods through a static library instead of relying on dynamically loaded them through a shared library is beneficial for two good reasons. Firstly, it greatly simplifies your application deployment as your application executable will have your native methods statically linked into it and, secondly, the security of your application improves. One potential issue with dynamic library loading is that a malicious person could replace your native method implementations with a shared library containing implementations that, instead of managing your data, might erase your hard disk, or worse!

To register native methods within your application, you should write your program as usual. That is, configure and create your JVM. However, before you locate the method IDs for your methods, you need to initialise an array containing information on the native methods to register and register them. The following stub of example code shows how to do this:

```
    ...

    /** Configure the JVM... */
    ...

    /** Create the JVM... */
#ifdef JNI_VERSION_1_2
    JNI_CreateJavaVM( &jvm, (void **)&env, &vmArgs );
#else
    JNI_CreateJavaVM( &jvm, &env, &vmArgs );
#endif /** JNI_VERSION_1_2 */

    /** Test that the JVM works by invoking a method in a test class */
    jclass testNativeClassClassblock = env->FindClass( "testNativeClass" );
    if ( testNativeClassClassblock == NULL ) {
        env->FatalError( "testNativeClassClassblock is NULL! Aborting!" );
      }

    /** Setup the native methods for registration */
    JNINativeMethod nativeMethods[] = {
        { "printString", "(Ljava/lang/String;)V",
          Java_testNativeClass_printString }
      };

    /** Register the native methods... */
    jint numRegistered =
        env->RegisterNatives( testNativeClassClassblock, nativeMethods, 1 );
    fprintf( stderr, "Successfully registered %d native methods\n", numRegis-
tered );
    if ( env->ExceptionOccurred() != NULL ) {
        env->ExceptionDescribe();
        exit( 1 );
        }

    ...
```

The description of the native methods is set up by creating an array of `JNINativeMethod` structures, one per native method to register. The elements of the structure are the name and the signature of the native method and the function pointer of your implementation of the native method. The function pointer is made accessible to your program by including the header file for the Java class containing the native method.

Once this structure has been declared, you simply need to call the `RegisterNatives()` JNI method. This associates the native methods from the JVM's point of view with your implementations. Therefore, once you invoke the native method, beit from your program or from Java, it'll work. And all without having to dynamically load the code!

It's perhaps worth mentioning a potential caveat in using `RegisterNatives()` that I tripped up on. If you declare your `JNINativeMethod` array before creating the JVM, `RegisterNatives()` will probably fail and, when the `NoSuchMethodError` triggered by the call is checked, it states that the class causing the problem is random text! Therefore, it's worth declaring your array of native method descriptors immediately before calling `RegisterNatives()` and to always check the return value is not negative.

Finally, we should discuss the issue of static compilation. When compiling the program using the Invocation API, we should simply additionally link the object file containing the native method implementations along with our program object files. Therefore, we'll simply end up with an application program with the native method implementations included. The example code including sample `Makefile`s and Visual Studio projects which should help you understand the process more clearly.

Embedding a JVM with RNI

Attaching To a JVM

The basic premise of using a JVM from within an existing application is that you *attach* the current thread of execution to the JVM such that RNI calls can be used throughout your application. These RNI calls will operate in exactly the same way as if you were using native methods from a Java application.

The whole issue of the loading and initialization of the JVM occurs when you attach the first thread using the `PrepareThreadForJava()` call. Therefore, there is no real explicit JVM startup sequence that you need to perform. The JVM will initialize itself when required.

The term "*thread*" is worth explaining here in order to understand exactly how your native application will interact with the JVM. A thread is a self-contained thread of execution within a program. In a standard native C or C++ program, at least one thread will exist namely the "main" thread of execution. If you have written a program that uses multi-threading, you will have several threads of execution within one process. This is fundamentally identical to the use of threading within Java.

In the simplest case, if you have a single-threaded program you should only need to attach to the JVM exactly once. That is, you only need to invoke `PrepareThreadForJava()` once.

This function is quite simple to use as it takes a single argument which is a pointer to a thread information structure of type `ThreadEntryFrame` and returns a boolean value indicating whether or not the thread attached successfully to the JVM.

For example, the simplest example we can usefully implement can be written as

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include "../../rnidefs.h"
#include "testClass.h"

int main( int argc, char **argv ) {

    /** Structure containing information about the attached thread */
    ThreadEntryFrame threadFrame;

    /** Class information for the "testClass" class */
    ClassClass *testClassClassblock;

    /** Attempt to attach the thread to the JVM */
    if ( PrepareThreadForJava( &threadFrame ) == TRUE ) {
        fprintf( stderr, "Attached thread to the JVM!\n" );

        /** Locate the classblock for "testClass" */
        testClassClassblock = FindClass( NULL, "testClass", 0 );
        if ( testClassClassblock == NULL ) {
            fprintf( stderr, "Failed to find class!\n" );
            exit( 1 );
          }

        /** Invoke the "printMessage()" method */
        execute_java_static_method( NULL, testClassClassblock, "printMes-
sage", "()V" );
      } else {
        fprintf( stderr, "Failed to attach the thread to Java\n" );
      }

    /** Detach the thread... */
    ...

    exit( 0 );
  }
```

When `PrepareThreadForJava()` is invoked for that thread, the thread information structure is populated with information pertinent to that attachment and should be stored for each attached thread as it will be required later when detaching from the JVM.

In applications using multiple-threads, *each thread* requiring interaction with the JVM *must* attach to the JVM prior to making any RNI calls otherwise undefined, but probably catastrophic, problems will occur. For example, if your application created a separate thread to read streaming data from the network, but the main thread exclusively handled the GUI, only the separate thread would need to call `PrepareThreadForJava()` as no RNI access is needed within the main program thread.

Detaching From a JVM

The corollary operation from attaching a thread to a JVM is to detach that thread once you have no further need to call back into the JVM from your native application. This operation should only ever be undertaken for threads that are already attached to the JVM.

Detaching a thread from a JVM is extremely easy and can be achieved by calling the RNI function called `UnprepareThreadForJava()`. This function takes a single argument which is the thread information structure for *that thread* populated when `PrepareThreadForJava()` was successfully executed. Using a thread information structure for a different thread is an extremely bad idea and can result in catastrophic errors in your program!

To illustrate the use of `UnprepareThreadForJava()`, we can re-implement the above example in the following code.

```
      /** We're finished with RNI calls, so detach the thread */
      UnprepareThreadForJava( &threadFrame );
```

Each thread attached to the JVM should have a corollary detach call to ensure that the JVM is not destabilised in any way. This is especially important if you are interacting with a JVM that is perhaps embedded within a device or within the operating system and is shared by multiple applications.

Notes

1. The "Java Virtual Machine" book published by O'Reilly should be consulted for more information on these topics.
2. This is not necessarily the case, but it is an extremely sensible idea to assume that it is. This ensures your code won't break in strange places.
3. Depending on the platform, this file may be buried quite deep into the hierarchy. For example, for Linux, the library is located in `$JAVA_HOME/lib/i586/green_threads`.
4. And, on Linux, the system `libpthread.so`.

Chapter 14. Native Rendering from Java

This chapter is designed to demonstrate some of the more practical uses for native methods programming in conjunction with integrating legacy code. It also discusses one of the most commonly asked questions I tend to get asked about native method programming and describes some of the problems inherent in this activity.

Possibly the most frequently asked question I get asked by native methods programmers is “how do I draw onto an AWT or Swing component from native

This capability has a myriad of uses in different types of applications ranging from 3-dimensional rendering from a high-performance rendering library such as *OpenGL* or *Direct3D*, realtime video streaming or even the display of documents in encoded formats such as *PDF*, *PostScript* or even *Microsoft Word*.

Establishing the how of doing this is simply a technological problem whereas establishing *why* you’d want to do this is a more thorny topic.

The main reason that drawing onto an existing Java component is useful is that you can treat the native window as you would any Java component. This functionality encompasses aspects of Java such as repainting partially or completely exposed windows and other event handling, such as mouse and keyboard events. The bonus of using Java for these tasks is that your window handling code will be totally portable. Therefore, all that is required from you is the underlying native code that provides data to draw onto the AWT or Swing component.

This might sound great. In fact, it is great. This functionality, coupled with either existing legacy libraries or new high-performance native code, can enable developers to deploy portable applications very quickly as any GUI integration code need only be written once. This speeds up the development time of multi-platform technologies quite dramatically and can lead to a far more stable and bug-free product.

Without this functionality, native code is constrained to creating its own “top-level” windows for drawing into and handling its own window-system specific events for window manipulation. This destroys the seamlessness of slotting, say, a realtime video stream into an video-conferencing application of which 90% of the code is written in Java.

There are three other good reasons for architecting your code to draw onto an AWT Component. The first reason is one we have discussed in the previous chapters namely, why rewrite all that legacy code that you know works? The second reason is *speed*. Much as the gap in execution speeds between Java and compiled code is closing steadily, Java will have a hard time beating the assembler-level hand-tuned optimizations put into existing software such as OpenGL or a realtime video decoder. For these applications, porting to Java would not only consume vast amounts of programmer-hours but it would also result in a far more sluggish system. In the cases of 3D graphics or video decoding, speed is everything. The final reason is that Java cannot take advantage of hardware accelerated operations, for example, a 3D graphics accelerator card. By using native code to perform your rendering, you can take full advantage of the performance a hardware accelerator can give you. This is true for both 3D graphics and other multimedia areas such as video playback.

However, there are some important caveats to these techniques, some political and others technical. From a political point of view, using native code to render onto Java is most certainly not “100% Pure Java”. The use of native methods in Java programs is generally frowned upon by the “Pure Java” proponents. This fact has a knock-on effect into the technical issues.

There are several technical issues surrounding drawing onto Java components from native code. Firstly, since native methods are not “Pure Java”, JavaSoft do not guarantee that the technique that we’re about to explore will continue to be supported in future releases of the Sun JVM. This stems from the political stance I mentioned in the previous paragraph. A further technical issue is that the way in which the internal data is accessed uses internal Java classes. These classes are bundled with the Sun VM and derivatives, but are regarded as being “subject to change” and should not be used directly by developers. A final point is that the way in which these classes are manipulated varies between JVMs and operating systems bringing up quite a few issues that must be clarified before true portability can occur.

The fact that the internal classes are subject to change is quite a worrying development. The way in which underlying AWT information can be accessed from native methods changed radically for the better between the release of JDK-1.0 and JDK-1.1 and above. Microsoft’s JVM uses its own proprietary access path which has remained stable throughout all their JVMs.

Inside AWT Components

AWT components are a group of functional GUI parts that can be snapped together to build complex and sophisticated user interfaces. The AWT is also completely window-system transparent and will translate the platform-independent AWT calls and components into appropriate window-system specific widgets.

The architecture of the AWT is built around the concept of peer classes. Each AWT component had a corresponding peer class which actually implemented the native method code that hooked into a particular window system. Thus, the classes defined in the `java.awt` package could be written completely in Java with the peers handling the platform specific functionality. This system worked perfectly well but consumed quite heavy quantities of memory to store internal structures for each of the native window-system widgets and was tied closely to the underlying window-system. The peer architecture was also quite difficult to extend and could only support components that had native code written for them. Therefore, components that have an associated peer class is generally known as being a “*heavyweight*” component.

Since heavyweight components are closely tied to the underlying window-system, each component has a native window into which it is drawn. This allows us to locate the information on the window-system resources that will enable us to “piggyback” our native rendering onto that window.

Given that we will be writing an application that performs 3D rendering onto a window, it makes the most sense to subclass the `java.awt.Canvas` class which you can use in Java to draw onto. This will allow us to slot our native rendering into an existing drawing framework with very little effort.

After all this hoo-hah, a good question still remains. What exactly is it that the peer class provides that is needed to allow rendering to the AWT component? Native code generally will require some sort of *window handle* to draw onto. In X Windows, this is usually called a *drawable* and is of data type `Window`. In Win32, a generic window handle of data type `HWND` represents the window.

The ways in which we extract the window handle depends on both the operating system or window system that the JVM is current running on as well as depending on the actual JVM implementation itself. That is to say, the technique used for the Sun JVM is not the same as that used with the Microsoft JVM. Similarly, the techniques differ slightly between UNIX and Win32 platforms.

The following sections will discuss how you can extract the window information from Java components in various JVMs and also how to use this information safely within a multi-threaded environment, such as Java.

Swing / JFC

To confuse matters further, the JDK-1.1 in advance of the new *Java Foundation Classes* or *Swing* API, have shifted focus from the old architecture of using *peer* classes to new "*lightweight*" components. This impacts the ability to access internal window system information in order to draw onto the Java component.

The main problem arises from the fact that where AWT components have a one-to-one mapping between themselves and native window system information, Swing components do not. Swing components are implemented purely within Java and use Java to draw themselves, not native code. This poses us a real problem in terms of finding a valid native window to render onto.

Swing does touch AWT in a few places, typically to create top-level windows into which other Swing components are drawn. Therefore, for a complex Swing GUI, there might be only one component with a peer class that can be rendered into.

To confound things further, Swing components and heavyweight components are generally not rendered at the same time which results in a particularly nasty problem known as the *z-ordering problem*. This occurs when a heavyweight component is placed into a Swing-based GUI and the resulting effect is that the heavyweight components will always hide any Swing components underneath!

Therefore, not only do we need to do a lot of extra work to find out the window information to render into, we also need to be aware of potential z-ordering problems.

Finally, if we do decide to draw onto a Swing component from native code, the offset of that component within its particular layout managers must be taken into account in order for the native code to render into the correct place. This is necessary because the target Swing component will be occupying the same peered window as many other Swing components. Therefore, if we simply render to the entire peer window instead of a defined region of that window, we'll overwrite all the other Swing components.

Therefore, there are two general solutions to rendering to a Swing component from native code. One is to take into account the fact that a peer class needs to be located within the GUI hierarchy, offsets need to be calculated to work out where to draw and z-ordering issues need to be addressed before actually rendering *directly* from native code to the peer window. The second approach is to render into a virtual framebuffer, or *offscreen buffer*, then use Java's standard `Graphics` class to copy that buffer to the Swing component.

Rendering Directly to Swing

If you decide to implement the first approach, the first thing you need to do is locate a peered component within the GUI component hierarchy. This can be achieved by traversing recursively up the hierarchy testing each component as you go until a heavyweight component is located. This is possible using the `isHeavyweight()` method defined within the core Swing `JComponent` class. For example

```
Component c = Swing canvas to render onto;
while ( JComponent.isLightweight( c ) ) {
    /** Recurse up the hierarchy */
    c = c.getParent();
  }
```

This code will give you the heavyweight component that the target Swing component is using as a peer and you can then use *this* component to extract the underlying window-system information as discussed in later sections.

The larger problem is to correctly track the position of the target Swing component within the peer window. That is, if we are rendering an image directly to a top-level window which contains a lot of Swing components, we must make sure that we render exactly the correct size of image to exactly the correct location, otherwise we'll cause havoc with the displaying of the other components.

This is theoretically simple to do in that the target Swing component can be asked for its location and dimensions. Unfortunately, the dimensions relate the the size of the component unaffected by components within which it is contained which may alter the visible region of the target component. For example, a scroller pane window will allow sub-regions of the target component to be visible without affecting the actual component dimensions.

Furthermore, because of the way that Swing is architected, the offset of the target component is always given as being the offset from its *parent* container and not the top-level window. Therefore, we need to recurse upwards through the component hierarchy again totally the offsets to determine where the native code has to render to.

Sound a bit complicated yet? It gets worse. This doesn't take into account things like internal frame title bars or customized borders around components. If your target component is sitting within one of these graphical entities, you need to make additional offset calculations to work out where to draw.

There is no straight-forward or surefire way to work this out although it is possible to implement a solution targetted completely for your application. This will make things easier, but still not completely straight-forward.

Rendering to a Virtual Framebuffer

Rendering to a *virtual framebuffer* and copying that to the Swing component can be a simpler way of integrating native rendering with Swing than direct rendering to a target Swing component.

The theory behind this technique is that the renderer renders its output to a virtual framebuffer, or basically a block of memory representing a window, then uses the standard Java `Graphics` class methods to transfer that buffer to the Swing component. Since Swing components are implemented purely in Java using the `Graphics` class, this technique slots quite nicely into the general scheme of things.

Furthermore, you completely sidestep issues such as z-ordering and offset tracking. Z-ordering is cured because you are rendering to the Swing component *via* Java as opposed to a native renderer and the offset of the Swing component is already implicit within the `Graphics` object associated with each Swing component.

This sounds great! All the benefits of the previous technique with none of the really horrible problems!

Not quite. This technique has several deficiencies, typically involving performance. The two main problem areas are firstly, because a virtual framebuffer is being used, hardware acceleration will not be used when rendering and secondly, because Java's image manipulation and transfer APIs are being used[1], the performance of the transfer from virtual framebuffer to Swing component may be quite poor. Therefore, if you decide to use this technique for native code rendering to a Swing component, be aware that the performance may be totally unacceptable.

Summing up, it is currently extremely difficult to implement native code rendering to a Swing component for various reasons. On one hand, you could implement the high-performance approach which requires a lot of work in order to correctly track the rendering within the Swing hierarchy. On the other hand, you can implement an easier approach which has potentially unacceptable performance. Neither way is particularly satisfactory, to be honest. Sun are aware of these problems and are working to solve them.

Locating Window Information

The first stage in rendering to an existing Java GUI component from native code is to locate the window-system information for the window to which we wish to render.

This process differs literally on a per-JVM and a per-platform basis, so it can become quite difficult to maintain over a large number of JVMs and platforms. This section simply discusses *how* to extract the information and a later section will discuss how to organize your window extraction code in a more elegant and maintainable manner.

For the sake of simplicity, I've separated JVMs into two groups, those that are derived or follow the Sun JVM implementation and the Microsoft JVM.

The SUN JVM and Derivatives

The Sun JVM is used by many vendors as a reference implementation to implement their their own JVM technologies from. This situation ensures that a good bulk of these technologies all support JNI and the techniques I'm about to tell you about. From your point of view, this ensures that you will get a reasonably high level of coverage across deployed JVMs on end-user's machines. It also means you don't need to maintain a horrendous number of source code branches to support all these virtual machines!

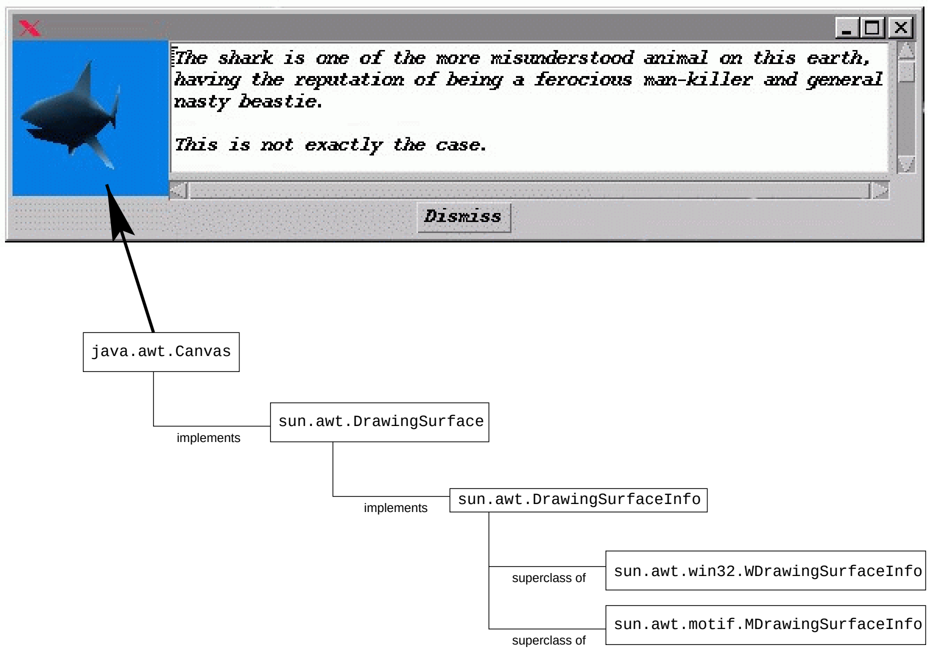
Drawing Surfaces

For the JDK-1.1 compliant virtual machine, Sun has engineered a new way in which underlying windows are referenced. These windows are now termed *drawing surfaces* and each AWT component that supports native window access has a drawing surface associated with it.

Information on the drawing surface associated with a given AWT class can be accessed *via* an interface called `DrawingSurfaceInfo` which describes a particular drawing surface. Each heavyweight class implements the `sun.awt.DrawingSurface` interface which defines only one method namely `getDrawingSurfaceInfo()`.

However, here's the crunch. `DrawingSurfaceInfo` is simply a generic interface that provides no real low-level information. To access the actual information on the drawing surface *for the current operating system*, you need to use either the `sun.awt.motif.MDrawingSurfaceInfo`, the `sun.awt.windows.WDrawingSurfaceInfo` class or the `sun.awt.macos.ADrawingSurfaceInfo` class on X Windows / Motif platforms, Win32 and MacOS respectively. Figure illustrates the flow of this code more clearly and the following code stub illustrates how you can access the low-level information of a `CustomCanvas` object which is a simple subclass of the `java.awt.Canvas` class.

Figure 14-1. Drawing Surfaces


The shark is one of the more misunderstood animal on this earth, having the reputation of being a ferocious man-killer and general nasty beastie.

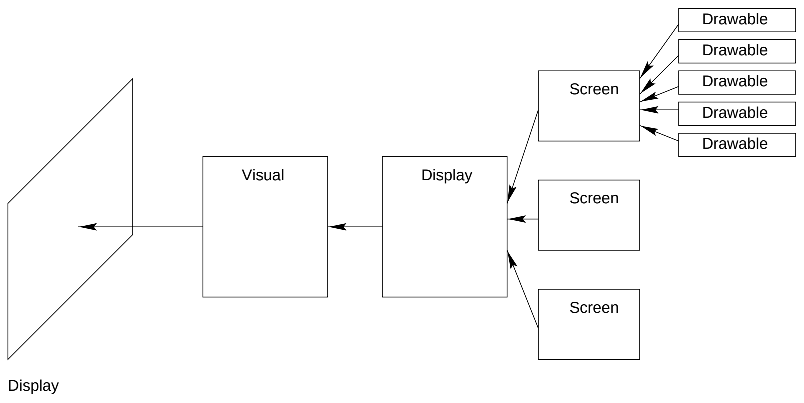
This is not exactly the case.

Dismiss
java.awt.Canvas
implements
sun.awt.DrawingSurface
implements
sun.awt.DrawingSurfaceInfo
superclass of
sun.awt.win32.WDrawingSurfaceInfo
superclass of
sun.awt.motif.MDrawingSurfaceInfo

```
/** Create a new drawing surface */
CustomCanvas c = new CustomCanvas();

/** Get the heavyweight peer object associated with this component */
ComponentPeer peer = c.getPeer();

/** Extract the drawing surface info for this surface via the peer */
DrawingSurfaceInfo dsi = ((DrawingSurface)peer).getDrawingSurfaceInfo();

/** Get hold of the OS we're using... */
String os = System.getProperty( "os.name" );

/** ...and extract the underlying surface info appropriately.. */
if ( os.startsWith( "Windows" ) ) {
    /** Convert the opaque DrawingSurfaceInfo object..... */
    sun.awt.windows.WDrawingSurfaceInfo wdsi =
        (sun.awt.windows.WDrawingSurfaceInfo)dsi;
    /** Now extract the HWND handle for this window */
    int hWND = wdsi.getHWnd();

    /** Do our custom processing using this window handle.... */
    ...
  } else {
    if ( os.startsWith( "MacOS" ) ) {
        /** Convert the opaque DrawingSurfaceInfo object..... */
        sun.awt.macos.ADrawingSurfaceInfo adsi =
            (sun.awt.macos.ADrawingSurfaceInfo)dsi;

        /** Now, extract the GWorld handle for this window... */
        int gWorld = adsi.getPort();

        /** Do our custom processing using this window handle... */
        ...
      } else {
        /** Assume we're running on a UNIX system using Motif... */
        /** Convert the opaque DrawingSurfaceInfo object... */
        sun.awt.motif.MDrawingSurfaceInfo mdsi =
            (sun.awt.motif.MDrawingSurfaceInfo)dsi;

        /** Now, extract the Drawable pointer for this window... */
        int drawable = mdsi.getDrawable();

        /** Do our custom processing using this window handle... */
        ...
      }
  }
```

Each window system returns slightly different information for the window handles. For example, the `sun.awt.motif.MDrawingSurfaceInfo` interface will return the X Window ID of the drawing surface when the `getDrawable()` method is invoked. Equally importantly, the ID of the X server display is also accessible *via* the `getDisplay()` method. Other items of potentially useful information such as the ID of the allocated colourmap and X visual can also be accessed *via* methods defined in this interface.

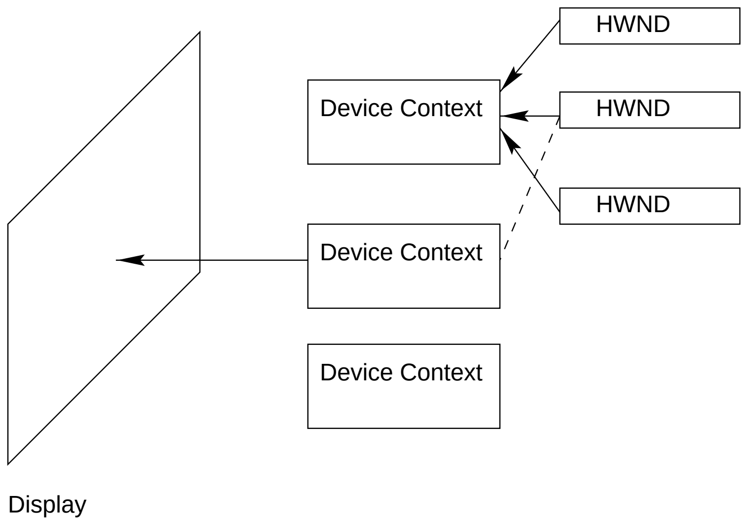
Figure 14-2. X11 Windowing Architecture



Similarly, the `sun.awt.win32.WDrawingSurfaceInfo` interface returns information on a window created under Win32. The methods defined in this interface are similar in function to those provided in the Motif interface but differ slightly in terms of the underlying window system data they provide.

For example, window IDs under Win32 are represented by an `HWND` value which is returned by the `getHwnd()` method. Furthermore, in Win32 to draw to a window, a structure known as a *device context* is sometimes required. This too can be accessed *via* `getHdc()`.

Figure 14-3. Win32 Windowing Architecture



These values, whatever the window system, represent the underlying window ID that is encapsulated by the `Canvas` class or another heavyweight component. Therefore, if you wished to draw onto an AWT component from native code, these values represent the drawing surfaces to draw onto.

Of course, if you look at the return types of these functions, they simply return `int` values not `HWND` values or `Display` values. This is a neat trick that can be done with Java regarding storing structures or memory areas allocated in native code within a Java class. Instead of storing the memory itself, you simply copy the *pointer* to that memory into a Java `int` value. Therefore, when you drop down into native code, you simply cast the `int` to the appropriate type, *i.e.*,

```
int drawablePointer = ...;    /** Value extracted by getDrawable() */
int displayPointer = ...;     /** Value extracted by getDisplay() */
...
/** Get the pointer to the X display */
Display *dpy = (Display *)displayPointer;

/** Get the pointer to the X window */
Window *drawable = (Window *)drawablePointer;
...
```

The pointers to the native window structures, after casting, can be used immediately to draw or render images onto the AWT component. However, a fully functional integration with Java components and native rendering is not quite as straight-forward as simply being able to draw. We also require the capability to handle events being generated against that window. In this case, we have two options for implementing code. The first option is to write native code event handlers which trap appropriate events directly and route them accordingly within your application or you could use Java's AWT event handlers and integrate that event handling with native methods.

In legacy applications with custom event handlers, the first route might seem initially attractive, but you must be aware that Java is a fully multi-threaded language and implements event handling on top of this multi-threaded core. In these circumstances, your own event handlers within native code can potentially cause problems if Java *also* tries to perform some operation on the same window.

These problems can be almost completely circumvented by *locking* the drawing surface prior to carrying out any operations on it.

Locking Drawing Surfaces

Java is a multi-threaded environment which brings its own set of problems to native method programming. This is more evident when integrating AWT, native methods and underlying window-system functions since the underlying window system may not be "*thread-safe*". This section explains how locking can be performed by you to circumvent multi-threading issues that may arise in your applications when integrating with AWT.

Thread-safeness is a term used to describe code that can be used within a multi-threaded environment without being subject to "interference" or corruption when being run simultaneously by multiple threads. Another term for functions that are thread-safe is that they are *re-entrant*. For example, code using and updating values stored within global variables are not thread-safe. The following example has a single global variable that can be updated from two functions that are driven from two threads. One function updates the value by one, the other by ten.

```
int value = 0;

void thread1() {
    while ( 1 ) {
        value++;
        printf( "thread1: value = %d\n", value );
      }
  }

void thread2() {
    while ( 1 ) {
        value += 10;
        printf( "thread2: value = %d\n", value );
      }
  }
```

Under normal circumstances, the desired effect would be that the value would print values like:

```
thread1: value = 0
thread2: value = 0
thread1: value = 1
thread2: value = 10
thread1: value = 2
thread2: value = 20
thread1: value = 3
thread2: value = 30
...
```

whereas in actual fact you are likely to end up with

```
thread1: value = 0
thread1: value = 1
thread2: value = 11
thread1: value = 21
thread1: value = 22
thread1: value = 23
thread2: value = 33
...
```

which is totally wrong. Since both threads are updating the same variable, each thread will corrupt the value for the other thread causing bizarre problems for the developer to track down. A good example of the problems that this can cause is the `strtok()` string handling function defined in ANSI C. This function stores a global pointer to track the position in a string. However, if `strtok()` is called from multiple threads, the pointer value will be moved to potentially illegal positions within the wrong string causing either program crashes or simply wrong results. The `errno` variable is another thread-unsafe aspect of the standard C libraries.

Window systems, such as X11, also suffer from these problems, where global variables are used heavily[2]. However, since Java is a multi-threaded environment this has the potential to cause problems.

Java provides “system threads” in its JVM to handle functionality that all users and programmers are likely to want. For example, a system thread for garbage collection exists as do threads to handle AWT functionality. The existence of these threads imply that the AWT is both asynchronous and multi-thread-aware whereas X11 is simply asynchronous. Therefore, the possiblity of AWT events or activity causing thread corruption in X11 is quite high but not predictable.

The nature of these problems occur through the exercising of a effect known as a *race condition*. This occurs when two resources compete for a single resource. Depending on which one wins the race for resource, different effects may manifest in your code. For example, in the example above concerning the global variables, if the two threads competed to update the value and the first thread won the first 6 races the program would appear to run correctly. However, when the second thread wins, the program starts behaving strangely and starts generating wrong answers.

This is quite important in windowing environments where operations take place asynchronously and also may take a period of time to execute. Both of these aspects can exercise race conditions in the strangest places providing you with false debugging information and useless stack traces. Therefore, a system needs to be implemented that removes the threat of race conditions from your code.

The internal peer classes that the AWT uses all perform *locking* which is implemented internally within the AWT. When any underlying window-system functionality is about to be used, the AWT will attempt to acquire a lock which regulates access to the window-system functions. If another AWT thread is using the window-system functions, the other thread will wait until the first thread releases that lock. This ensures that only one thread can ever access the window-system functions at the same time and circumvents the lack of thread-safety in these window systems.

Since you will be implementing code using the underlying window system and integrating this with AWT peer classes, you will need to use the locking mechanisms that the `sun.awt.DrawingSurfaceInfo` classes provide.

There are two methods in the `sun.awt.DrawingSurfaceInfo` classes that are used for locking. These are named `lock()` and `unlock()`. The `lock()` method attempts to acquire the global AWT lock required to allow you to call underlying window system functions. Similarly, you should *always* call `unlock()` after you are finished otherwise AWT will deadlock completely. It is also good programming practice to lock as little code as you need otherwise performance of your application may suffer. That is, in most cases it is only necessary to protect one function call that will use the underlying window system. If the native method performs additional processing before or after the window-system call, you shouldn't lock this code at all since it will hold the lock much longer than is actually necessary. When this happens, AWT itself will be waiting on the lock being released which might cause sluggish user response times.

Under X11, you might see an error appear from the X server along the lines of:

```
Xlib: unexpected async reply (sequence 0x68fe)!
```

upon which your application locks up completely. This error occurs when the X server receives input from two threads simultaneously causing a corruption internally in the X server. This is exactly the error that explicitly locking and unlocking your peer class is designed to circumvent.

You may have noticed that X11 is discussed quite heavily in regard with problems caused by multi-threading. Win32 platforms do not seem to be affected by these issues since Win32 is inherently thread-safe, but it is still prudent to explicitly lock and unlock your drawing surfaces rather than take the chance of a stray event mangling your application.

A important regarding locking is that 3rd party libraries or legacy code that you might be interfacing with in your native methods may also internally call window-system specific functions. A good example of this is the OpenGL function `glViewport()` which is usually called when the window containing an OpenGL window is resized. This function resizes the various buffers that OpenGL uses to render into and will call window-system specific functions internally. Since `glViewport()` is an expensive operation, there is a very good chance that AWT will attempt to asynchronously process another event during a `glViewport()` call. Under X11, this will likely cause an error of the “unexpected async reply” variety locking up your application. Therefore, to avoid this problem you should call `lock()` prior to `glViewport()` and `unlock()` immediately afterwards. This will ensure that X does not trip up.

There is a caveat to locking *via* the drawing surface classes. If you need to lock a native method against AWT interference when a drawing surface is *not* involved, then this technique will not work. You have to resort to slightly more underhand methods.

For example, you might wish to use an OpenGL offscreen rendering buffer. In this scenario, the buffer is created independent of any drawing surfaces but is rendered to from native code and may use certain window-system calls to copy data around. Without locking, there is a very real possibility that an AWT call might execute simultaneously with an offscreen buffer operation. When this occurs, the application will lock up completely.

A neat technique to circumvent this issue is to explicitly lock and unlock using the AWT monitor that all AWT methods must acquire prior to performing any window-system operations.

Under Java-2 and above, this can be achieved by declaring a variable within your code corresponding to the object used with the JVM for the AWT monitor. This should be declared as:

```
extern jobject awt_lock;
```

Locking and unlocking can now simply be achieved by calling the standard JNI `MonitorEnter()` and `MonitorExit()` functions using the `awt_lock` variable as the object to lock on.

Avoiding these problems is extremely important since an error of this sort will cause your application to unrecoverably lock up. There are no warnings, no exceptions and no second chances!

The AWT Native Interface

Another way in which you can locate window information has been introduced in JDK-1.3 called the "AWT Native Interface". This is a collection of functions that allow you to interface with AWT components directly from native code through a specifically designed API, as opposed to our slightly sneaky back-door technique.

The core of this API is implemented through native code reflections of the AWT DrawingSurface classes that we have already discussed. As with the sneaky method, this AWT exposes the functionality critical to manipulating AWT components such as acquiring and releasing drawing surfaces and locking and unlocking those drawing surfaces prior to and after use.

For example, a simple drawing implementation using either X11 or Win32 might be written as

```
#include <jawt_md.h>

JNIEXPORT void JNICALL
Java_CustomCanvas_nPaint( JNIEnv *env, jobject arg ) {
    JAWT awt;
    JAWT_DrawingSurface *ds;
    JAWT_DrawingSurfaceInfo *dsi;
#ifdef HAVE_LIBX11
    JAWT_X11DrawingSurfaceInfo *dsi;
#else
    JAWT_Win32DrawingSurfaceInfo *dsi;
#endif

    /** The AWT version must be set prior to using the API */
    awt.version = JAWT_VERSION_1_3;
    if ( JAWT_GetAWT( env, &awt ) == JNI_FALSE ) {
        fprintf( stderr, "Cannot locate AWT!\n" );
        return;
      }

    /** Get the drawing surface for the component */
    ds = awt.GetDrawingSurface( env, arg );

    /** Lock the drawing surface */
    ds->Lock( ds );

    /** If failed to acquire the lock, release the drawing surface and re-
turn */
    if ( lock & JAWT_LOCK_ERROR ) != 0 ) {
        fprintf( stderr, "Failed to lock drawing surface\n" );
        awt.FreeDrawingSurface( ds );
        return;
      }

    /** Get the drawing surface info */
#ifdef HAVE_LIBX11
    dsi = (JAWT_X11DrawingSurfaceInfo *)dsi->platformInfo;
#else
```

```
    dsi = (JAWT_Win32DrawingSurfaceInfo *)dsi->platformInfo;
#endif

    /** Do our drawing through the window-system structures in ’dsi’ */
    ...

    /** Free the drawing surface info */
    ds->FreeDrawingSurfaceInfo( dsi );

    /** Unlock the drawing surface */
    ds->Unlock( ds );

    /** Free the drawing surface */
    awt.FreeDrawingSurface( ds );

    return;
  }
```

The important things to note about this interface are, firstly, that each drawing surface info structure contains platform-dependent information and must be conditionally compiled for each platform and, secondly, that if you acquire a lock on a drawing surface, you *must* remember to release it! If you fail to do so, your application will hang the AWT!

The Microsoft JVM

The Microsoft JVM was originally based on the Sun reference JVM implementation. However, over the years it has mutated into being an extremely powerful Win32-specific technology capable of executing Java bytecode at an astonishing rate due to optimized garbage collection and excellent JIT capabilities.

Microsoft realized before Sun that programmers might wish to access the underlying window that an AWT component encapsulates and provided a standard[3] mechanism to do this.

The Microsoft JVM has defined a Microsoft-specific interface which most AWT component classes implements named `com.ms.awt.peer.ComponentPeerX`. This interface defines several methods which allow you to locate window handles for Win32 windows. The method that is of most use for rendering into an AWT component from native code is `getHwnd()` which returns the `HWND` value for the given AWT component. And that’s literally all there is to it!

For example, if you have a class that extends `java.awt.Canvas`, you can invoke `getHwnd()` against an object instantiated from that class immediately. There are no additional layers of indirection to support platform independence. A piece of Java code to extract the `HWND` value from a given class is

```
/** Our component class that is rendered onto from native code */
CustomCanvas dragc = new CustomCanvas( 200, 200 );

/** Extract the pointer to the HWND of this component */
int hwndPointer = dragc.getHwnd();

/** Do some native processing */
dragc.createNativeWindow( hwndPointer );
```

This value can then be passed into native code as an integer and cast to the `HWND` data type for use in the native Win32 functions for window manipulation. For example

```
RNIEXPORT void RNICALL
CustomCanvas_createNativeWindow( struct HCustomCanvas *canvas,
                                 long hwndPointer ) {

    /** Cast the extracted value to HWND */
    HWND windowHandle = (HWND)hwndPointer;

    /** Do some Win32-specific stuff */
    ...
  }
```

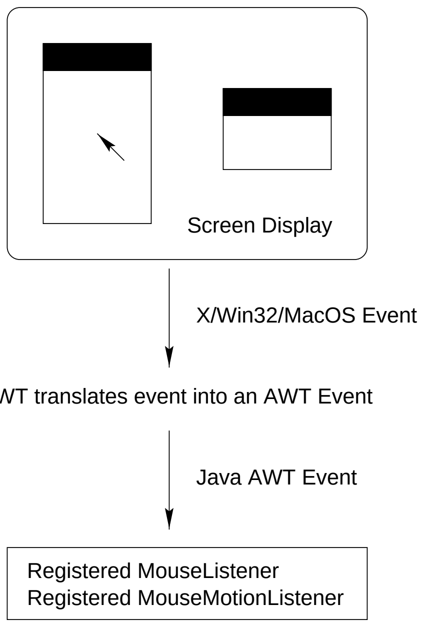
Although this extracting the window handle is extremely easy, there are some major problems that you might encounter when implementing code in the Microsoft JVM. The most common occurrence of these issues arises from the interaction between multi-threading and window handling which I shall discuss later in this chapter.

Integrating Native Rendering and AWT

The previous sections of this chapters discussed the techniques that can be applied to the commoner JVMs in order to support rendering or drawing to an AWT or Swing component from native code. The other aspect, and the aspect that will be visible to programmers wishing to use this functionality, is to integrate this native rendering capability into the AWT way of doing things such that Java applications can use it easily.

The AWT is designed around the principle of components and listeners. That is, each AWT component, such as a `Canvas`, can have listeners attached to it through which events of a certain type are routed. For example, if you wished to track the position of the mouse whilst it is over a particular AWT component, you would register a `MouseMotionListener` against that component. Therefore, whenever the underlying window system picks up an event signifying that the mouse has moved, a Java AWT event is routed into Java for you to query. Figure illustrates the concept.

Figure 14-4. AWT Event Routing



Using these event listeners seems like a good thing to do. They encourage structured event handling, they are completely portable and, from my personal point of view, turn event handling code into something that isn't complete gibberish. You can actually follow what's supposed to be happening.

Therefore, by rendering directly onto an AWT component from native code, we automatically are blessed with the ability to use the event listener mechanisms present in AWT straight away.

The most important functionality that graphical components generally require is the ability to refresh or repair damage to the window either from hiding and showing the window or the window appearing in the first place. Similarly, this functionality usually doubles up to redraw the screen when the data in your application is to be drawn. For example, the application will request a screen refresh whenever something moves in the world.

In X Windows you are required to track *Exposure events* to allow for window repainting. These events are generated whenever a window is "brought-forward" from behind another window, or part of the window needs redrawn for whatever reason. Not only are you required to trap these events, but you also need to parse them for the appropriate redraw information to correctly refresh the window, or part of a window, that has been exposed. This sounds pretty painful and, believe me, it can be. Can we improve on this by using AWT's event mechanism?

Of course we can. Each AWT component uses two methods called `repaint()` and `paint()` to regulate component redrawing when exposed. Because of the way that AWT sits on top of the underlying window system, whenever an exposure event is trapped, AWT simply calls the `paint()` or `repaint()` methods for the appropriate components.

From the native rendering perspective, this is ideal in that we only need one entry point to the underlying native code to handle screen redrawing. This native method simply needs to be called within a custom `paint()` and `repaint()` method in order to completely satisfy our needs.

Putting these different pieces of the jigsaw together, we can write a fairly straight-forward AWT component that allows native rendering to be performed in the following way.

```
import java.awt.*;

public class CustomCanvas extends Canvas {

    /** Native method that performs initialization of the native window */
    private boolean nInitialize();

    /** A value used to store the pointer to the underlying window data */
    private int pData = -1;

    /** Native method used to repaint the window when required */
    private void nRepaint();

    /** Overridden painting methods */
    public void paint( Graphics g ) {
        /** Re-render the scene and display it */
        nRepaint();
      }

    /** Repaints the component */
    public void repaint() {
        /** Re-render the scene and display it */
        nRepaint();
      }
  }
```

This component would have the corresponding native implementations of the `nInitialize()` and `nRepaint()` methods implemented using one of the techniques discussed earlier in this chapter.

This general framework will work portably across X-based, Win32, MacOS and OS/2 using JNI and Win32 using Microsoft's JVM. The worst part by far is dealing with the extraction of the window information, but once that stage has been accomplished, manipulation of the low-level windows is actually quite straight-forward as the AWT has taken most of the pain of event handling away.

The benefits of this approach are fairly immediate in that your Java code will now be totally portable and make GUI development far easier since the majority of horrors is now hidding "under the hood". This can only be a good thing.

Summing Up

This chapter has only begun to scratch the surface of the uses and power inherent in integrating custom AWT components native rendering code. There are many pitfalls, more than I could possibly hope to list and elucidate in this book. The number of pitfalls increases exponentially as you migrate your code to different JVMs and different operating systems, but my personal view is that it's worth the effort. The relief in not having to write explicit Motif or Win32 event handlers is beyond measure!

This chapter has also concentrated on rendering to native windows. I have not specified any particular way in which this rendering is to occur but merely left it blank. This blank is where your legacy code and applications comes in. The techniques outlined in this chapter will provide you with a window to draw onto but what you draw is entirely up to you. It could be 3D graphics *via* OpenGL or a Word document, PDF, PostScript, real-time video...the list is endless.

All of the above are possible and you too can reap the benefits of Java's flexibility and ease of use coupled with the raw performance that native code and possible hardware acceleration can give you. Not to mention deploy some awesomely powerful applications that'll impress your users no end!

Notes

1. Such as `ImageProducer` and `ImageConsumer`.
2. X11R6 is now thread-safe on many popular UNIX platforms. You can use a function called `XInitThreads()` to check whether your X libraries and X server have been built in a thread-safe configuration. Solaris is known to be thread-safe, whereas Linux generally is not. Linux, however, can be re-built to be thread-safe using the GNU `glibc` and LinuxThreads. This task is outwith the scope of this book.
3. By standard, I mean that it exists in both versions 1 and 2 of the Microsoft JVM. Whether or not it will continue to be supported remains to be seen. However, Microsoft's proprietary J/Direct technology also seems to use this functionality internally which bodes well for continuing support.

Chapter 15. JNI Function Reference

This chapter contains quick reference material for the JNI functions. It covers Version 1.2 of the JNI, which is the current version as of Java 2 SDK Version 1.3. Collections of related functions that vary only by the type of data on which they operate appear under special parametric headings. For example, all the functions that call non-static methods are listed under “Call<Type>Method”.

AllocObject

Name

`AllocObject` —

Synopsis

```
jobject AllocObject( JNIEnv *env, jclass clazz )
```

Parameters

`env`

The native interface pointer

`clazz`

The class information of the class to be instantiated

Description

Allocates a new object of the given class, but does not call any of the object’s constructors

AttachCurrentThread

Name

`AttachCurrentThread` —

Synopsis

```
jint AttachCurrentThread( JavaVM *vm, void **penv, jint version )
```

Parameters

`vm`

The JVM to attach to

`penv`

The native interface pointer

`version`

The required version of the JNI function table

Description

Attaches the current thread of execution within the native program to the specified JVM. This allows you to interface directly with the JVM from that thread.

Call<Type>Method

Name

`Call<Type>Method` —

Synopsis

```
jboolean CallBooleanMethod( JNIEnv *env, jobject obj, jmethodID methodID,
                            args )
jbyte CallByteMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
jchar CallCharMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
jdouble CallDoubleMethod( JNIEnv *env, jobject obj, jmethodID metho-
dID, args )
jfloat CallFloatMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
jint CallIntMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
jlong CallLongMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
jobject CallObjectMethod( JNIEnv *env, jobject obj, jmethodID methodID,
                          args )
jshort CallShortMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
void CallVoidMethod( JNIEnv *env, jobject obj, jmethodID methodID, args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in comma-separated form

Description

`CallBooleanMethod()`, `CallByteMethod()`, `CallCharMethod()`, `CallDoubleMethod()`, `CallFloatMethod()`, `CallIntMethod()`, `CallLongMethod()`, `CallObjectMethod()`, `CallShortMethod()`, and `CallVoidMethod()` all invoke the specified Java instance method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in a comma-separated list.

Call<Type>MethodA

Name

`Call<Type>MethodA` —

Synopsis

```
jboolean CallBooleanMethodA( JNIEnv *env, jobject obj,
                             jmethodID methodID, jvalue *args )
jbyte CallByteMethodA( JNIEnv *env, jobject obj,
                       jmethodID methodID, jvalue *args )
jchar CallCharMethodA( JNIEnv *env, jobject obj,
                       jmethodID methodID, jvalue *args )
jdouble CallDoubleMethodA( JNIEnv *env, jobject obj,
                           jmethodID methodID, jvalue *args )
jfloat CallFloatMethodA( JNIEnv *env, jobject obj,
                         jmethodID methodID, jvalue *args )
jint CallIntMethodA( JNIEnv *env, jobject obj,
                     jmethodID methodID, jvalue *args )
jlong CallLongMethodA( JNIEnv *env, jobject obj,
                       jmethodID methodID, jvalue *args )
jobject CallObjectMethodA( JNIEnv *env, jobject obj,
                           jmethodID methodID, jvalue *args )
jshort CallShortMethodA( JNIEnv *env, jobject obj,
                         jmethodID methodID, jvalue *args )
void CallVoidMethodA( JNIEnv *env, jobject obj,
                      jmethodID methodID, jvalue *args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified as an array of `jvalues`

Description

`CallBooleanMethodA()`, `CallByteMethodA()`, `CallCharMethodA()`, `CallDoubleMethodA()`, `CallFloatMethodA()`, `CallIntMethodA()`, `CallLongMethod()A`, `CallObjectMethodA()`, `CallShortMethodA()`, and `CallVoidMethodA()` all invoke the specified Java instance method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied as an array of `jvalues`.

Call<Type>MethodV

Name

`Call<Type>MethodV` —

Synopsis

```
jboolean CallBooleanMethodV( JNIEnv *env, jobject obj,
                             jmethodID methodID, va_list args )
jbyte CallByteMethodV( JNIEnv *env, jobject obj,
                       jmethodID methodID, va_list args )
jchar CallCharMethodV( JNIEnv *env, jobject obj,
                       jmethodID methodID, va_list args )
jdouble CallDoubleMethodV( JNIEnv *env, jobject obj,
                           jmethodID methodID, va_list args )
jfloat CallFloatMethodV( JNIEnv *env, jobject obj,
                         jmethodID methodID, va_list args )
jint CallIntMethodV( JNIEnv *env, jobject obj,
                     jmethodID methodID, va_list args )
jlong CallLongMethodV( JNIEnv *env, jobject obj,
                       jmethodID methodID, va_list args )
jobject CallObjectMethodV( JNIEnv *env, jobject obj,
                           jmethodID methodID, va_list args )
jshort CallShortMethodV( JNIEnv *env, jobject obj,
                         jmethodID methodID, va_list args )
void CallVoidMethodV( JNIEnv *env, jobject obj,
                      jmethodID methodID, va_list args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in a varargs list

Description

`CallBooleanMethodV()`, `CallByteMethodV()`, `CallCharMethodV()`, `CallDoubleMethodV()`, `CallFloatMethodV()`, `CallIntMethodV()`, `CallLongMethodV()`, `CallObjectMethodV()`, `CallShortMethodV()`, and `CallVoidMethodV()` all invoke the specified Java instance method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in the form of an ANSI varargs list.

CallNonvirtual<Type>Method

Name

`CallNonvirtual<Type>Method` —

Synopsis

```
jboolean CallNonvirtualBooleanMethod( JNIEnv *env, jobject obj, jclass clazz,
                                      jmethodID methodID, args )
jbyte CallNonvirtualByteMethod( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, args )
jchar CallNonvirtualCharMethod( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, args )
jdouble CallNonvirtualDoubleMethod( JNIEnv *env, jobject obj, jclass clazz,
                                    jmethodID methodID, args )
jfloat CallNonvirtualFloatMethod( JNIEnv *env, jobject obj, jclass clazz,
                                  jmethodID methodID, args )
jint CallNonvirtualIntMethod( JNIEnv *env, jobject obj, jclass clazz,
                              jmethodID methodID, args )
jlong CallNonvirtualLongMethod( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, args )
jobject CallNonvirtualObjectMethod( JNIEnv *env, jobject obj, jclass clazz,
                                    jmethodID methodID, args )
jshort CallNonvirtualShortMethod( JNIEnv *env, jobject obj, jclass clazz,
                                  jmethodID methodID, args )
void CallNonvirtualVoidMethod( JNIEnv *env, jobject obj, jclass clazz,
                               jmethodID methodID, args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method

`clazz`

The class in which to actually invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in the comma-separated form

Description

`CallNonvirtualBooleanMethod()`, `CallNonvirtualByteMethod()`, `CallNonvirtualCharMethod()`, `CallNonvirtualDoubleMethod()`, `CallNonvirtualFloatMethod()`, `CallNonvirtualIntMethod()`, `CallNonvirtualLongMethod()`, `CallNonvirtualObjectMethod()`, `CallNonvirtualShortMethod()`, and `CallNonvirtualVoidMethod()` all invoke the specified Java method with the indicated return type in the given Java class on the given Java object. The specified class must be a superclass of the class of the specified Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in a comma-separated list.

CallNonvirtual<Type>MethodA

Name

`CallNonvirtual<Type>MethodA` —

Synopsis

```
jboolean CallNonvirtualBooleanMethodA( JNIEnv *env, job-
ject obj, jclass clazz,
                                       jmethodID methodID, jvalue *args )
jbyte CallNonvirtualByteMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                 jmethodID methodID, jvalue *args )
jchar CallNonvirtualCharMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                 jmethodID methodID, jvalue *args )
jdouble CallNonvirtualDoubleMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                     jmethodID methodID, jvalue *args )
jfloat CallNonvirtualFloatMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                   jmethodID methodID, jvalue *args )
jint CallNonvirtualIntMethodA( JNIEnv *env, jobject obj, jclass clazz,
                               jmethodID methodID, jvalue *args )
jlong CallNonvirtualLongMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                 jmethodID methodID, jvalue *args )
jobject CallNonvirtualObjectMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                     jmethodID methodID, jvalue *args )
jshort CallNonvirtualShortMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                   jmethodID methodID, jvalue *args )
void CallNonvirtualVoidMethodA( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, jvalue *args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method `clazz`

The class in which to actually invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in an array of `jvalues`

Description

`CallNonvirtualBooleanMethodA()`, `CallNonvirtualByteMethodA()`, `CallNonvirtualCharMethodA()`, `CallNonvirtualDoubleMethodA()`, `CallNonvirtualFloatMethodA()`, `CallNonvirtualIntMethodA()`, `CallNonvirtualLongMethodA()`, `CallNonvirtualObjectMethodA()`, `CallNonvirtualShortMethodA()`, and `CallNonvirtualVoidMethodA()` all invoke the specified Java method with the indicated return type in the given Java class on the given Java object. The specified class must be a superclass of the class of the specified Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied as an array of `jvalues`.

CallNonvirtual<Type>MethodV

Name

`CallNonvirtual<Type>MethodV` —

Synopsis

```
jboolean CallNonvirtualBooleanMethodV( JNIEnv *env, job-
ject obj, jclass clazz,
                                       jmethodID methodID, va_list args )
jbyte CallNonvirtualByteMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, va_list args )
jchar CallNonvirtualCharMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, va_list args )
jdouble CallNonvirtualDoubleMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                    jmethodID methodID, va_list args )
jfloat CallNonvirtualFloatMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                  jmethodID methodID, va_list args )
jint CallNonvirtualIntMethodV( JNIEnv *env, jobject obj, jclass clazz,
                              jmethodID methodID, va_list args )
jlong CallNonvirtualLongMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                jmethodID methodID, va_list args )
jobject CallNonvirtualObjectMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                    jmethodID methodID, va_list args )
jshort CallNonvirtualShortMethodV( JNIEnv *env, jobject obj, jclass clazz,
                                  jmethodID methodID, va_list args )
void CallNonvirtualVoidMethodV( JNIEnv *env, jobject obj, jclass clazz,
                               jmethodID methodID, va_list args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java object against which to invoke the method

`clazz`

The class in which to actually invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in a varargs list

Description

`CallNonvirtualBooleanMethodV()`, `CallNonvirtualByteMethodV()`, `CallNonvirtualCharMethodV()`, `CallNonvirtualDoubleMethodV()`, `CallNonvirtualFloatMethodV()`, `CallNonvirtualIntMethodV()`, `CallNonvirtualLongMethodV()`, `CallNonvirtualObjectMethodV()`, `CallNonvirtualShortMethodV()`, and `CallNonvirtualVoidMethodV()` all invoke the specified Java method with the indicated return type in the given Java class on the given Java object. The specified class must be a superclass of the class of the specified Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in an ANSI varargs list.

CallStatic<Type>Method

Name

`CallStatic<Type>Method` —

Synopsis

```
jboolean CallStaticBooleanMethod( JNIEnv *env, jclass clazz,
                                  jmethodID methodID, args )
jbyte CallStaticByteMethod( JNIEnv *env, jclass clazz,
                            jmethodID methodID, args )
jchar CallStaticCharMethod( JNIEnv *env, jclass clazz,
                            jmethodID methodID, args )
jdouble CallStaticDoubleMethod( JNIEnv *env, jclass clazz,
                                jmethodID methodID, args )
jfloat CallStaticFloatMethod( JNIEnv *env, jclass clazz,
                              jmethodID methodID, args )
jint CallStaticIntMethod( JNIEnv *env, jclass clazz,
                          jmethodID methodID, args )
jlong CallStaticLongMethod( JNIEnv *env, jclass clazz,
                            jmethodID methodID, args )
jobject CallStaticObjectMethod( JNIEnv *env, jclass clazz,
                                jmethodID methodID, args )
jshort CallStaticShortMethod( JNIEnv *env, jclass clazz,
                              jmethodID methodID, args )
void CallStaticVoidMethod( JNIEnv *env, jclass clazz,
                           jmethodID methodID, args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java class against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in comma-separated form

Description

`CallStaticBooleanMethod()`, `CallStaticByteMethod()`, `CallStaticCharMethod()`, `CallStaticDoubleMethod()`, `CallStaticFloatMethod()`, `CallStaticIntMethod()`, `CallStaticLongMethod()`, `CallStaticObjectMethod()`, `CallStaticShortMethod()`, and `CallStaticVoidMethod()` all invoke the specified Java class (`static`) method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in a comma-separated list.

CallStatic<Type>MethodA

Name

`CallStatic<Type>MethodA` —

Synopsis

```
jboolean CallStaticBooleanMethodA( JNIEnv *env, jclass clazz,
                                   jmethodID methodID, jvalue *args )
jbyte CallStaticByteMethodA( JNIEnv *env, jclass clazz,
                             jmethodID methodID, jvalue *args )
jchar CallStaticCharMethodA( JNIEnv *env, jclass clazz,
                             jmethodID methodID, jvalue *args )
jdouble CallStaticDoubleMethodA( JNIEnv *env, jclass clazz,
                                 jmethodID methodID, jvalue *args )
jfloat CallStaticFloatMethodA( JNIEnv *env, jclass clazz,
                               jmethodID methodID, jvalue *args )
jint CallStaticIntMethodA( JNIEnv *env, jclass clazz,
                           jmethodID methodID, jvalue *args )
jlong CallStaticLongMethodA( JNIEnv *env, jclass clazz,
                             jmethodID methodID, jvalue *args )
jobject CallStaticObjectMethodA( JNIEnv *env, jclass clazz,
                                 jmethodID methodID, jvalue *args )
jshort CallStaticShortMethodA( JNIEnv *env, jclass clazz,
                               jmethodID methodID, jvalue *args )
void CallStaticVoidMethodA( JNIEnv *env, jclass clazz,
                            jmethodID methodID, jvalue *args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java class against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in an array of `jvalues`

Description

`CallStaticBooleanMethodA()`, `CallStaticByteMethodA()`, `CallStaticCharMethodA()`, `CallStaticDoubleMethodA()`, `CallStaticFloatMethodA()`, `CallStaticIntMethodA()`, `CallStaticLongMethodA()`, `CallStaticObjectMethodA()`, `CallStaticShortMethodA()`, and `CallStaticVoidMethodA()` all invoke the specified Java class (`static`) method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied as an array of `jvalues`.

CallStatic<Type>MethodV

Name

`CallStatic<Type>MethodV` —

Synopsis

```
jboolean CallStaticBooleanMethodV( JNIEnv *env, jclass clazz,
                                   jmethodID methodID, va_list args )
jbyte CallStaticByteMethodV( JNIEnv *env, jclass clazz,
                             jmethodID methodID, va_list args )
jchar CallStaticCharMethodV( JNIEnv *env, jclass clazz,
                             jmethodID methodID, va_list args )
jdouble CallStaticDoubleMethodV( JNIEnv *env, jclass clazz,
                                 jmethodID methodID, va_list args )
jfloat CallStaticFloatMethodV( JNIEnv *env, jclass clazz,
                               jmethodID methodID, va_list args )
jint CallStaticIntMethodV( JNIEnv *env, jclass clazz,
                           jmethodID methodID, va_list args )
jlong CallStaticLongMethodV( JNIEnv *env, jclass clazz,
                             jmethodID methodID, va_list args )
jobject CallStaticObjectMethodV( JNIEnv *env, jclass clazz,
                                 jmethodID methodID, va_list args )
jshort CallStaticShortMethodV( JNIEnv *env, jclass clazz,
                               jmethodID methodID, va_list args )
void CallStaticVoidMethodV( JNIEnv *env, jclass clazz,
                            jmethodID methodID, va_list args )
```

```
jchar CallStaticCharMethod( JNIEnv *env, jclass clazz,
                            jmethodID methodID, args )
jchar CallStaticCharMethodA( JNIEnv *env, jclass clazz,
                             jmethodID methodID, yjvalue *args )
jchar CallStaticCharMethodV( JNIEnv *env, jclass clazz,
                             jmethodID methodID, va_list args )
```

Parameters

`env`

The native interface pointer

`obj`

The Java class against which to invoke the method

`methodID`

The unique identifier for a Java method

`args`

The correct number of arguments for the method, specified in a varargs list

Description

`CallStaticBooleanMethodV()`, `CallStaticByteMethodV()`, `CallStaticCharMethodV()`, `CallStaticDoubleMethodV()`, `CallStaticFloatMethodV()`, `CallStaticIntMethodV()`, `CallStaticLongMethodV()`, `CallStaticObjectMethodV()`, `CallStaticShortMethodV()`, and `CallStaticVoidMethodV()` all invoke the specified Java class (`static`) method with the indicated return type on the given Java object. The result of the method is returned as a value of the appropriate type. The method is specified by a unique method identifier previously fetched with `GetMethodID()`. Each argument that the method expects should be supplied in an ANSI varargs list.

DefineClass

Name

`DefineClass` —

Synopsis

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

Parameters

`env`

The native interface pointer

`name`

The fully-qualified name of the class to create

`loader`

A `ClassLoader` object

`buf`

An allocated buffer that contains the raw byte codes that define the class

`bufLen`

The size of the buffer

Description

Defines a new Java class from raw byte codes.

DeleteGlobalRef

Name

`DeleteGlobalRef` —

Synopsis

```
void DeleteGlobalRef( JNIEnv *env, jobject globalRef )
```

Parameters

`env`

The native interface pointer

`globalRef`

The previously created global reference to delete

Description

Deletes a global reference that has been created for an object.

DeleteLocalRef

Name

`DeleteLocalRef` —

Synopsis

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

Parameters

`env`

The native interface pointer

`localRef`

The local reference, or Java object, to garbage collect

Description

Marks an object as ready for garbage collection and causes it to be removed from memory as soon as possible.

DeleteWeakGlobalRef

Name

`DeleteWeakGlobalRef` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`obj`

The weak global reference

Description

Deletes the given weak global reference.

DestroyJavaVM

Name

`DestroyJavaVM` —

Synopsis

```
jint DestroyJavaVM( JavaVM *vm )
```

Parameters

`vm`

The JVM to destroy

Description

Destroys the specified Java Virtual Machine.

DetachCurrentThread

Name

`DetachCurrentThread` —

Synopsis

```
jint DetachCurrentThread( JavaVM *vm )
```

Parameters

`vm`

The JVM to detach from

Description

Detaches a previously attached thread from the given JVM. After this function has executed, you cannot interface with the JVM from this thread until you reattach to the JVM.

EnsureLocalCapacity

Name

`EnsureLocalCapacity` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`capacity`

The minimum number of required local references

Description

Indicate to the developer whether or not a potentially large number of local references can be allocated within the current native method prior to attempting creation of these references. For example, if you want to create 1,000,000 local references within a native method temporarily, the JVM might not support this and crash. By calling `EnsureLocalCapacity()`, you can find out in advance whether this allocation will succeed. Returns zero upon success or a negative number upon failure. An `OutOfMemoryError` is also thrown upon failure.

ExceptionCheck

Name

`ExceptionCheck` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

Description

Returns `JNI_TRUE` when there is a pending exception or `JNI_FALSE` if no exceptions are pending.

ExceptionClear

Name

`ExceptionClear` —

Synopsis

```
void ExceptionClear( JNIEnv *env )
```

Parameters

`env`

The native interface pointer

Description

Clears any pending `Exception` objects from the exception stack.

ExceptionDescribe

Name

`ExceptionDescribe` —

Synopsis

```
void ExceptionDescribe( JNIEnv *env )
```

Parameters

`env`

The native interface pointer

Description

Essentially invokes `printStackTrace()` on the given `Exception` object. The output is written to `System.err`, or the equivalent, in the current JVM (e.g., the Java Console in Netscape Navigator).

ExceptionOccurred

Name

`ExceptionOccurred` —

Synopsis

```
jthrowable ExceptionOccurred( JNIEnv *env )
```

Parameters

`env`

The native interface pointer

Description

Determines whether an `Exception` is pending. If so, a reference to the `Exception` object is returned.

FatalError

Name

`FatalError` —

Synopsis

```
void FatalError( JNIEnv *env, const char *msg )
```

Parameters

`env`

The native interface pointer

`msg`

The message to display with the JVM shutdown

Description

Terminates the JVM immediately and irrevocably.

FindClass

Name

`FindClass` —

Synopsis

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

Parameters

`env`

The native interface pointer

`name`

The fully-qualified name of the class to locate

Description

Locates information about the given class. `FindClass()` only operates on classes located in your `CLASSPATH` environment variable.

FromReflectedField

Name

`FromReflectedField` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`field`

A reference to a `java.lang.reflect.Field` object

Description

Returns the field ID that corresponds to the specified `java.lang.reflect.Field` or `NULL` if an exception occurs.

FromReflectedMethod

Name

`FromReflectedMethod` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`field`

A reference to a `java.lang.reflect.Method` object

Description

Returns the method ID that corresponds to the specified `java.lang.reflect.Method` or `NULL` if an exception occurs.

Get<Type>ArrayElements

Name

`Get<Type>ArrayElements` —

Synopsis

```
jboolean *GetBooleanArrayElements( JNIEnv *env, jbooleanArray array,
                                   jboolean *isCopy )
jbyte *GetByteArrayElements( JNIEnv *env, jbyteArray array,
                             jboolean *isCopy )
jchar *GetCharArrayElements( JNIEnv *env, jcharArray array,
                             jboolean *isCopy )
jdouble *GetDoubleArrayElements( JNIEnv *env, jdoubleArray array,
                                 jboolean *isCopy )
jfloat *GetFloatArrayElements( JNIEnv *env, jfloatArray array,
                               jboolean *isCopy )
jint *GetIntArrayElements( JNIEnv *env, jintArray array,
                           jboolean *isCopy )
jlong *GetLongArrayElements( JNIEnv *env, jlongArray array,
                             jboolean *isCopy )
jshort *GetShortArrayElements( JNIEnv *env, jshortArray array,
                               jboolean *isCopy )
```

Parameters

`env`

The native interface pointer

`array`

The array from which to extract the elements

`isCopy`

The address of a variable that indicates whether the extracted elements are a direct pointer to or copy of the original array elements

Description

`GetBooleanArrayElements()`, `GetByteArrayElements()`, `GetCharArrayElements()`, `GetDoubleArrayElements()`, `GetFloatArrayElements()`, `GetIntArrayElements()`, `GetLongArrayElements()`, and `GetShortArrayElements()` each extracts all of the elements in the specified array into a contiguous C array.

Get<Type>ArrayRegion

Name

`Get<Type>ArrayRegion` —

Synopsis

```
void GetBooleanArrayRegion( JNIEnv *env, jbooleanArray array, jsize start,
                            jsize len, jboolean *buf )
void GetByteArrayRegion( JNIEnv *env, jbyteArray array, jsize start,
                         jsize len, jbyte *buf )
void GetCharArrayRegion( JNIEnv *env, jcharArray array, jsize start,
                         jsize len, jchar *buf )
void GetDoubleArrayRegion( JNIEnv *env, jdoubleArray array, jsize start,
                           jsize len, jdouble *buf )
void GetFloatArrayRegion( JNIEnv *env, jfloatArray array, jsize start,
                          jsize len, jfloat *buf )
void GetIntArrayRegion( JNIEnv *env, jintArray array, jsize start,
                        jsize len, jint *buf )
void GetLongArrayRegion( JNIEnv *env, jlongArray array, jsize start,
                         jsize len, jlong *buf )
void GetShortArrayRegion( JNIEnv *env, jshortArray array, jsize start,
                          jsize len, jshort *buf )
```

Parameters

`env`

The native interface pointer

`array`

The array from which to extract a region

`start`

The starting element of the region

`len`

The number of elements to be extracted

`buf`

A pre-allocated C array large enough to contain extracted elements

Description

`GetBooleanArrayRegion()`, `GetByteArrayRegion()`, `GetCharArrayRegion()`, `GetDoubleArrayRegion()`, `GetFloatArrayRegion()`, `GetIntArrayRegion()`, `GetLongArrayRegion()`, and `GetShortArrayRegion()` each extracts a contiguous portion of the specified array.

Get<Type>Field

Name

`Get<Type>Field` —

Synopsis

```
jboolean GetBooleanField( JNIEnv *env, jobject obj, jfieldID fieldID )
jbyte GetByteField( JNIEnv *env, jobject obj, jfieldID fieldID )
```

```
jchar GetCharField( JNIEnv *env, jobject obj, jfieldID fieldID )
jdouble GetDoubleField( JNIEnv *env, jobject obj, jfieldID fieldID )
jfloat GetFloatField( JNIEnv *env, jobject obj, jfieldID fieldID )
jint GetIntField( JNIEnv *env, jobject obj, jfieldID fieldID )
jlong GetLongField( JNIEnv *env, jobject obj, jfieldID fieldID )
jobject GetObjectField( JNIEnv *env, jobject obj, jfieldID fieldID )
jshort GetShortField( JNIEnv *env, jobject obj, jfieldID fieldID )
```

Parameters

`env`

The native interface pointer

`obj`

The object from which to extract a field value

`fieldID`

The unique identifier for the desired field

Description

`GetBooleanField()`, `GetByteField()`, `GetCharField()`, `GetDoubleField()`, `GetFloatField()`, `GetIntField()`, `GetLongField()`, `GetObjectField()`, and `GetShortField()` each returns the value contained in an instance field of the speficied type within the given object.

GetArrayLength

Name

`GetArrayLength` —

Synopsis

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

Parameters

`env`

The native interface pointer

`array`

The array to be counted

Description

Returns the number of elements within the given array.

GetEnv

Name

`GetEnv` —

Synopsis

```
jint GetEnv( JavaVM *vm, void **penv, jint version )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`vm`

A pointer to a JVM

`penv`

A native interface pointer

`version`

A version number

Description

If the current thread is attached to the JVM, `penv` is set to the appropriate native interface pointer and the function returns `JNI_OK`. If the current thread is not attached to the JVM, `penv` is set to `NULL` and the function returns `JNI_EDETACHED`. If the specified version is not supported (an earlier JVM, for instance), `penv` is set to `NULL` and the function returns `JNI_EVERSION`. in native code.

GetFieldID

Name

`GetFieldID` —

Synopsis

```
jfieldID GetFieldID( JNIEnv *env, jclass clazz,
                     const char *name, const char *sig )
```

Parameters

`env`

The native interface pointer

`clazz`

The class in which to look for a field

`name`

The name of the field

`sig`

The signature of the field

Description

Locates a unique identifier for an instance field within a Java class. This identifier allows you to perform operations on that field in native code.

GetJavaVM

Name

`GetJavaVM` —

Synopsis

```
jint GetJavaVM( JNIEnv *env, JavaVM **vm )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`vm`

A pointer to a location to store the JVM

Description

Returns a pointer to the JVM to which the current thread is attached.

GetMethodID

Name

`GetMethodID` —

Synopsis

```
jmethodID GetMethodID( JNIEnv *env, jclass clazz,
                       const char *name, const char *sig )
```

Parameters

`env`

The native interface pointer

`clazz`

The class in which to look for a method

`name`

The name of the method

`sig`

The signature of the method

Description

Locates a unique identifier for an instance method within a Java class. This identifier allows you to invoke the Java method from native code.

GetObjectArrayElement

Name

`GetObjectArrayElement` —

Synopsis

```
jobject GetObjectArrayElement( JNIEnv *env, jobjectArray array,
                               jsize index )
```

Parameters

`env`

The native interface pointer

`array`

The array from which to fetch an element `index`

The index of the desired element

Description

Extracts a single Java `Object` from an array of objects.

GetObjectClass

Name

`GetObjectClass` —

Synopsis

```
jclass GetObjectClass( JNIEnv *env, jobject obj )
```

Parameters

`env`

The native interface pointer

`obj`

The object from with to retrieve class information

Description

Returns class information for the specified object.

GetPrimitiveArrayCritical

Name

`GetPrimitiveArrayCritical` —

Synopsis

```
void GetPrimitiveArrayCritical( JNIEnv *env, jarray array,
                                jboolean *isCopy )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`array`

The Java array

`isCopy`

The address of a variable that indicates whether the extracted elements are a direct pointer to or copy of the original array elements

Description

This function is generally identical to the standard `Get<Type>ArrayElements()` functions, but calling it increases the likelihood that a pointer to the array elements is returned instead of a copy of the elements. This call begins a “critical section” of code. Inside this critical section, the currently executing thread must not block and no other JNI functions can be executed. To exit the critical section, call

`ReleasePrimitiveArrayCritical`. `GetPrimitiveArrayCritical()` calls can be nested, however, and should be freed in the correct order.

GetStatic<Type>Field

Name

`GetStatic<Type>Field` —

Synopsis

```
jboolean GetStaticBooleanField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jbyte GetStaticByteField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jchar GetStaticCharField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jdouble GetStaticDoubleField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jfloat GetStaticFloatField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jint GetStaticIntField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jlong GetStaticLongField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jobject GetStaticObjectField( JNIEnv *env, jclass clazz, jfieldID fieldID )
jshort GetStaticShortField( JNIEnv *env, jclass clazz, jfieldID fieldID )
```

Parameters

`env`

The native interface pointer

`clazz`

The class from which to extract a field value

`fieldID`

The unique identifier for the desired field

Description

`GetStaticBooleanField()`, `GetStaticByteField()`, `GetStaticCharField()`, `GetStaticDoubleField()`, `GetStaticFloatField()`, `GetStaticIntField()`, `GetStaticLongField()`, `GetStaticObjectField()`, and `GetStaticShortField()` each returns the value contained in a class (`static`) field of the speficied type within the given object.

GetStaticFieldID

Name

`GetStaticFieldID` —

Synopsis

```
jfieldID GetStaticFieldID( JNIEnv *env, jclass clazz,
                           const char *name, const char *sig )
```

Parameters

`env`

The native interface pointer

`clazz`

The class in which to look for a field

`name`

The name of the field

`sig`

The signature of the field

Description

Locates a unique identifier for a class (`static`) field within a Java class. This identifier allows you to perform operations on that field in native code.

GetStaticMethodID

Name

`GetStaticMethodID` —

Synopsis

```
jmethodID GetStaticMethodID( JNIEnv *env, jclass clazz,
                             const char *name, const char *sig )
```

Parameters

`env`

The native interface pointer

`clazz`

The class in which to look for a method

`name`

The name of the method

`sig`

The signature of the method

Description

Locates a unique identifier for a class (`static`) method within a Java class. This identifier allows you to invoke the Java method from native code.

GetStringChars

Name

`GetStringChars` —

Synopsis

```
const jchar *GetStringChars( JNIEnv *env, jstring string,
                             jboolean *isCopy )
```

Parameters

`env`

The native interface pointer

`string`

The Java `String`

`isCopy`

The address of a variable that indicates whether the extracted characters are a direct pointer to or copy of the original array elements

Description

Extracts the characters from the Java `String` in Unicode format.

GetStringCritical

Name

`GetStringCritical` —

Synopsis

```
const jchar *GetStringCritical( JNIEnv *env, jstring string,
                                jboolean *isCopy )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`string`

The Java `String`

`isCopy`

The address of a variable that indicates whether the extracted characters are a direct pointer to or copy of the original array elements

Description

This function is generally identical to the `GetStringChars()` function, but calling it increases the likelihood that a pointer to the characters is returned instead of a copy of the elements. This call begins a “critical section” of code. Inside this critical section, the currently executing thread must not block and no other JNI functions can be executed. To exit the critical section, call `ReleaseStringCritical`. `GetStringCritical()` calls can be nested, however, and should be freed in the correct order.

GetStringLength

Name

`GetStringLength` —

Synopsis

```
jsize GetStringLength( JNIEnv *env, jstring string )
```

Parameters

`env`

The native interface pointer

`string`

The Java `String`

Description

Calculates the length of a given Java `String` in terms of Unicode characters.

GetStringRegion

Name

`GetStringRegion` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`str`

The Java `String`

`start`

The starting element within the string

`len`

The number of bytes to be copied

`buf`

The buffer that will contain the string region

Description

Extracts the given substring of a Java `String` to a standard C array of Unicode characters.

GetStringUTFChars

Name

`GetStringUTFChars` —

Synopsis

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

Parameters

`env`

The native interface pointer

`string`

The Java `String`

`isCopy`

The address of a variable that indicates whether the extracted characters are a direct pointer to or copy of the original array elements

Description

Extracts the characters from the Java `String` in UTF-8 format (which is compatible with ASCII and ISO Latin-1).

GetStringUTFLength

Name

`GetStringUTFLength` —

Synopsis

```
jsize GetStringUTFLength( JNIEnv *env, jstring string )
```

Parameters

`env`

The native interface pointer

`string`

The Java `String`

Description

Calculates the length of a given Java `String` in terms of UTF-8 characters (which are compatible with ASCII and ISO Latin-1).

GetStringUTFRegion

Name

`GetStringUTFRegion` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`str`

The Java `String`

`start`

The starting element within the string

`len`

The number of bytes to be copied

`buf`

The buffer that will contain the string region

Description

Extracts the given substring of a Java `String` to a standard C array of UTF-8 characters (which are compatible with ASCII and ISO Latin-1).

GetSuperclass

Name

`GetSuperclass` —

Synopsis

```
jclass GetSuperclass( JNIEnv *env, jclass clazz )
```

Parameters

`env`

The native interface pointer

`clazz`

The class for which to retrieve superclass information

Description

Returns class information for the superclass of the given class.

GetVersion

Name

`GetVersion` —

Synopsis

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

Parameters

`env`

The native interface pointer

Description

Returns the version of JNI as implemented by the current JVM.

IsAssignableFrom

Name

`IsAssignableFrom` —

Synopsis

```
jboolean IsAssignableFrom( JNIEnv *env, jclass clazz1, jclass clazz2 )
```

Parameters

`env`

The native interface pointer

`clazz1`

The class information for the first class `clazz2`

The class information for the second class

Description

Tests whether an object instantiated from `clazz1` can be cast to an object of type `clazz2`.

IsInstanceOf

Name

`IsInstanceOf` —

Synopsis

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

Parameters

`env`

The native interface pointer

`obj`

The object to test

`clazz`

The class to test against

Description

Carries out the same operation as the `instanceof` keyword in Java code. The given object is tested to see if it is an instance of the given class.

IsSameObject

Name

`IsSameObject` —

Synopsis

```
jboolean IsSameObject( JNIEnv *env, jobject ref1, jobject ref2 )
```

Parameters

`env`

The native interface pointer

`ref1`

The first object

`ref2`

The second object

Description

Tests to see whether the two given objects are, in fact, identical.

JNI_CreateJavaVM

Name

`JNI_CreateJavaVM` —

Synopsis

```
jint JNI_CreateJavaVM( JavaVM **pvm, void **penv, void *args )
```

Parameters

`pvm`

A pointer to a location to store the new JVM

`penv`

A native interface pointer

`args`

An array of arguments

Description

Creates a new Java Virtual Machine for use within existing native programs. The newly created JVM is placed within the address specified in the first argument. The JNI interface pointer is placed within the address specified by the second argument. The final argument points to an array of initialization arguments that are used to regulate the creation of the JVM.

JNI_GetCreatedJavaVMs

Name

`JNI_GetCreatedJavaVMs` —

Synopsis

```
jint JNI_GetCreatedJavaVMs( JavaVM **vmBuf, jsize bufLen, jsize *nVMs )
```

Parameters

`vmBuf`

An array of created JVMs

`bufLen`

Maximum number of entries to be written to the buffer

`nVMs`

The number of created JVMs

Description

Returns a buffer of the created JVMs within the currently executing program. A pointer to each JVM is written to the given buffer and the actual number of JVMs present is stored in the `nVMs` argument.

JNI_GetDefaultJavaVMInitArgs

Name

`JNI_GetDefaultJavaVMInitArgs` —

Synopsis

```
void JNI_GetDefaultJavaVMInitArgs( void *vm_args )
```

Availability

Deprecated as of Java 2 SDK Version 1.2

Parameters

`vm_args`

The default JVM initialization arguments

Description

Populates the given argument with the default JVM initialization arguments required when creating a new JVM via the Invocation API. This function is deprecated as of Java 2 SDK Version 1.2, as the syntax of `JNI_CreateJavaVM()` has changed.

JNI_OnLoad

Name

`JNI_OnLoad` —

Synopsis

```
jint JNI_OnLoad( JavaVM *vm, void *reserved )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`vm`

A pointer to a Java Virtual Machine

`reserved`

Reserved for future use

Description

The JVM calls `JNI_OnLoad()` when the native library is loaded. `JNI_OnLoad()` must return the JNI version required by the native library to run correctly. For example, to use any of the new Java 2 SDK Version 1.2 JNI functions, `JNI_OnLoad()` must return `0x00010002`. Similarly, if `JNI_OnLoad()` is not defined within a native library, the JVM assumes that only Java 1.1 JNI functions are available. If the version number returned by `JNI_OnLoad()` is unrecognized or otherwise illegal, the library load fails completely.

JNI_OnUnload

Name

`JNI_OnUnload` —

Synopsis

```
void JNI_OnUnload( JavaVM *vm, void *reserved )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`vm`

A pointer to a Java Virtual Machine

`reserved`

Reserved for future use

Description

The JVM call `JNI_OnUnload()` when the class loader that loaded the native library is garbage collected. Therefore, `JNI_OnUnload()` can be used to perform clean up operations.

MonitorEnter

Name

`MonitorEnter` —

Synopsis

```
jint MonitorEnter( JNIEnv *env, jobject obj )
```

Parameters

`env`

The native interface pointer

`obj`

The object to be locked

Description

Enters the monitor associated with the specified object.

MonitorExit

Name

`MonitorExit` —

Synopsis

```
jint MonitorExit( JNIEnv *env, jobject obj )
```

Parameters

`env`

The native interface pointer

`obj`

The object to be unlocked

Description

Exits the monitor associated with the specified object.

New<Type>Array

Name

`New<Type>Array` —

Synopsis

```
jbooleanArray NewBooleanArray( JNIEnv *env, jsize length )
jbyteArray NewByteArray( JNIEnv *env, jsize length )
jcharArray NewCharArray( JNIEnv *env, jsize length )
jdoubleArray NewDoubleArray( JNIEnv *env, jsize length )
jfloatArray NewFloatArray( JNIEnv *env, jsize length )
jintArray NewIntArray( JNIEnv *env, jsize length )
jlongArray NewLongArray( JNIEnv *env, jsize length )
jshortArray NewShortArray( JNIEnv *env, jsize length )
```

Parameters

`env`

The native interface pointer

`length`

The length of the array

Description

`NewBooleanArray()`, `NewByteArray()`, `NewCharArray()`, `NewDoubleArray()`, `NewFloatArray()`, `NewIntArray()`, `NewLongArray()`, and `NewShortArray()`, each creates a new array of primitive values that is `length` elements long.

NewGlobalRef

Name

`NewGlobalRef` —

Synopsis

```
jobject NewGlobalRef( JNIEnv *env, jobject obj )
```

Parameters

`env`

The native interface pointer

`obj`

The object for which a global reference is to be made

Description

Creates a global reference for a given object, thereby eliminating the possibility that the object can be garbage collected.

NewLocalRef

Name

`NewLocalRef` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`ref`

The object for which to create a new reference

Description

Creates a new local reference to the specified reference and returns the new reference. The specified reference can be global or local.

NewObject, NewObjectA, NewObjectV

Name

`NewObject, NewObjectA, NewObjectV` —

Synopsis

```
jobject NewObject( JNIEnv *env, jclass clazz, jmethodID methodID, args )
jobject NewObjectA( JNIEnv *env, jclass clazz, jmethodID methodID,
                    jvalue *args )
jobject NewObjectV( JNIEnv *env, jclass clazz, jmethodID methodID,
                    va_list args )
```

Parameters

`env`

The native interface pointer

`clazz`

The class information of the class to be instantiated

`methodID`

The unique method identifier for the constructor to be invoked

`args`

The correct number of arguments for the constructor, in the format expected by the JNI function being used (i.e., comma-separated, an array of `jvalues`, or a varargs list).

Description

`NewObject()` allocates a new object of the desired class and invokes the specified constructor. Arguments to the constructor are passed as a comma-separated list of values.

`NewObjectA()` operates in a similar way to `NewObject()`, but requires that the arguments are specified as an array of `jvalues`.

`NewObjectV()` operates in a similar way to `NewObject()`, but requires that the arguments are specified as an ANSI varargs list.

NewObjectArray

Name

`NewObjectArray` —

Synopsis

```
jarray NewObjectArray( JNIEnv *env, jsize length, jclass elementClass,
                       jobject initialElement )
```

Parameters

`env`

The native interface pointer

`length`

The length of the array

`elementClass`

The class type of the array

`initialElement`

A Java object that specifies the default element value for the array

Description

Creates a new array of Java objects of the specified class and of the given dimension.

NewString

Name

`NewString` —

Synopsis

```
jstring NewString( JNIEnv *env, const jchar *bytes, jsize len )
```

Parameters

`env`
: The native interface pointer

`unicodeChars`
: A string of Unicode character data

`len`
: The number of Unicode characters in the string

Description

Creates a new Java `String` object from the given Unicode string.

NewStringUTF

Name

`NewStringUTF` —

Synopsis

```
jstring NewStringUTF( JNIEnv *env, const jchar *bytes, jsize len )
```

Parameters

`env`

The native interface pointer

`unicodeChars`

A string of UTF-8 character data

`len`

The number of UTF-8 characters in the string

Description

Creates a new Java `String` object from the given UTF-8 string (which is compatible with ASCII and ISO Latin-1).

NewWeakGlobalRef

Name

`NewWeakGlobalRef` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`obj`

The object for which to create a new weak global reference

Description

Reates a new weak global reference to the given object. If the object is `NULL` or if the JVM has no more memory available, `NULL` is returned and an `OutOfMemoryError` is thrown.

PopLocalFrame

Name

`PopLocalFrame` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`result`

An object for which to return a local reference or `NULL`

Description

Destroys the current local reference frame and deallocate all local references created within it. If `result` is not `NULL`, a local reference to the given object within the previous local reference frame is returned. If you do not need to return a local reference from the frame being destroyed, the `result` parameter should be specified as `NULL`.

PushLocalFrame

Name

`PushLocalFrame` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`capacity`

The number of local references to allocate

Description

Creates a new local frame in which the given number of local references can be created. `PushLocalFrame()` returns zero upon succesfull creation of a new frame and a negative number upon failure. An `OutOfMemoryError` is also thrown upon failure.

RegisterNatives

Name

`RegisterNatives` —

Synopsis

```
jint RegisterNatives( JNIEnv *env, jclass clazz,
                      const JNINativeMethod *methods, jint nMethods )
```

Parameters

`env`

The native interface pointer

`clazz`

The class with which the methods are to be associated

`methods`

An array of native methods to be registered

`nMethods`

The number of native methods to be registered

Description

Registers the specified native methods with the JVM and associates them with the specified class. The `methods` parameters specifies an array of `JNINativeMethods` structures that contain the names, signatures, and function pointers of the native methods, while `nMethods` indicates the number of native methods in the array.

Release<Type>ArrayElements

Name

`Release<Type>ArrayElements` —

Synopsis

```
void ReleaseBooleanArrayElements( JNIEnv *env, jbooleanArray array,
                                  jboolean *elements, jint mode )
void ReleaseByteArrayElements( JNIEnv *env, jbyteArray array,
                               jbyte *elements, jint mode )
void ReleaseCharArrayElements( JNIEnv *env, jcharArray array,
                               jchar *elements, jint mode )
void ReleaseDoubleArrayElements( JNIEnv *env, jdoubleArray array,
                                 jdouble *elements, jint mode )
void ReleaseFloatArrayElements( JNIEnv *env, jfloatArray array,
                                jfloat *elements, jint mode )
void ReleaseIntArrayElements( JNIEnv *env, jintArray array,
                              jint *elements, jint mode )
void ReleaseLongArrayElements( JNIEnv *env, jlongArray array,
                               jlong *elements, jint mode )
void ReleaseShortArrayElements( JNIEnv *env, jshortArray array,
                                jshort *elements, jint mode )
```

Parameters

`env`

The native interface pointer

`array`

The array from which the elements were extracted

`elements`

The previously extracted elements of the array `mode`

The mode used to free the array elements

Description

`ReleaseBooleanArrayElements()`, `ReleaseByteArrayElements()`, `ReleaseCharArrayElements()`, `ReleaseDoubleArrayElements()`, `ReleaseFloatArrayElements()`, `ReleaseIntArrayElements()`, `ReleaseLongArrayElements()`, and `ReleaseShortArrayElements()` each synchronizes the Java array and extracted array elements. A `mode` value of `0` causes the array elements to be written back to the Java array and the C array deallocated. A `mode` value of `JNI_COMMIT` copies the array elements back into the Java array but does not free the C array. A `mode` value of `JNI_ABORT` frees the C array but does not copy the possibly altered array elements back into the Java array.

ReleasePrimitiveArrayCritical

Name

`ReleasePrimitiveArrayCritical` —

Synopsis

```
void ReleasePrimitiveArrayCritical( JNIEnv *env, jarray array,
                                    void *carray, jint mode )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`array`

The Java array

`carray`

The array elements

`mode`

The mode used to free the array elements

Description

This function corresponds to `Release<Type>ArrayElements()`, but is used when you have used `GetPrimitiveArrayCritical()` to extract the elements of an array. Calling `ReleasePrimitiveArrayCritical()` marks the end of the "critical section" begun by calling `GetPrimitiveArrayCritical()`.

ReleaseStringChars

Name

`ReleaseStringChars` —

Synopsis

```
void ReleaseStringChars( JNIEnv *env, jstring string, const jchar *chars )
```

Parameters

`env`

The native interface pointer

`string`

The Java `String` from which the contents were extracted

`chars`

The buffer into which the `String` contents were extracted

Description

Copies the extracted Unicode characters forming the string back into the Java `String`.

ReleaseStringCritical

Name

`ReleaseStringCritical` —

Synopsis

```
void ReleaseStringCritical( JNIEnv *env, jstring string,
                            const jchar *carray )
```

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`string`

The Java `String`

`carray`

The extracted characters

Description

Releases the extracted string data and also marks the end of the critical section started when `GetStringCritical()` was invoked.

ReleaseStringUTFChars

Name

`ReleaseStringUTFChars` —

Synopsis

```
void ReleaseStringUTFChars( JNIEnv *env, jstring string,
                            const jbyte *chars )
```

Parameters

`env`

The native interface pointer

`string`

The Java `String` from which the contents were extracted

`chars`

The buffer into which the `String` contents were extracted

Description

Copies the extracted UTF-8 characters (compatible with ASCII and ISO Latin-1) forming the string back into the Java `String`.

Set<Type>ArrayRegion

Name

`Set<Type>ArrayRegion` —

Synopsis

```
void SetBooleanArrayRegion( JNIEnv *env, jbooleanArray array, jsize start,
                            jsize len, jboolean *buf )
void SetByteArrayRegion( JNIEnv *env, jbyteArray array, jsize start,
                         jsize len, jbyte *buf )
void SetCharArrayRegion( JNIEnv *env, jcharArray array, jsize start,
                         jsize len, jchar *buf )
void SetDoubleArrayRegion( JNIEnv *env, jdoubleArray array, jsize start,
                           jsize len, jdouble *buf )
void SetFloatArrayRegion( JNIEnv *env, jfloatArray array, jsize start,
                          jsize len, jfloat *buf )
void SetIntArrayRegion( JNIEnv *env, jintArray array, jsize start,
                        jsize len, jint *buf )
void SetLongArrayRegion( JNIEnv *env, jlongArray array, jsize start,
                         jsize len, jlong *buf )
void SetShortArrayRegion( JNIEnv *env, jshortArray array, jsize start,
                          jsize len, jshort *buf )
```

Parameters

`env`

The native interface pointer

`array`

The array in which the region of elements is to be set

`start`

The starting index of the region

`len`

The number of elements in the region

`buf`

A C array that contains the new element values

Description

`SetBooleanArrayRegion()`, `SetByteArrayRegion()`, `SetCharArrayRegion()`, `SetDoubleArrayRegion()`, `SetFloatArrayRegion()`, `SetIntArrayRegion()`, `SetLongArrayRegion()`, and `SetShortArrayRegion()` each copies the C buffer back into the designated region of the Java array.

Set<Type>Field

Name

`Set<Type>Field` —

Synopsis

```
void SetBooleanField( JNIEnv *env, jobject obj, jfieldID fieldID,
                      jboolean newValue )
```

```
void SetByteField( JNIEnv *env, jobject obj, jfieldID fieldID,
                   jbyte newValue )
void SetCharField( JNIEnv *env, jobject obj, jfieldID fieldID,
                   jchar newValue )
void SetDoubleField( JNIEnv *env, jobject obj, jfieldID fieldID,
                     jdouble newValue )
void SetFloatField( JNIEnv *env, jobject obj, jfieldID fieldID,
                    jfloat newValue )
void SetIntField( JNIEnv *env, jobject obj, jfieldID fieldID,
                  jint newValue )
void SetLongField( JNIEnv *env, jobject obj, jfieldID fieldID,
                   jlong newValue )
void SetObjectField( JNIEnv *env, jobject obj, jfieldID fieldID,
                     jobject newValue )
void SetShortField( JNIEnv *env, jobject obj, jfieldID fieldID,
                    jshort newValue )
```

Parameters

`env`

The native interface pointer

`obj`

The object in which to set the field value

`fieldID`

A unique identifier for the desired field

`newValue`

The value to which the field should be set

Description

`SetBooleanField()`, `SetByteField()`, `SetCharField()`, `SetDoubleField()`, `SetFloatField()`, `SetIntField()`, `SetLongField()`, `SetObjectField()`, and `SetShortField()` each sets the value of an instance field of the appropriate Java type to a new value within the given object.

SetObjectArrayElement

Name

`SetObjectArrayElement` —

Synopsis

```
void SetObjectArrayElement( JNIEnv *env, jobjectArray array,
                            jsize index, jobject value )
```

Parameters

`env`

The native interface pointer

`array`

The array in which to set an element

`index`

The index for the array element

`value`

The new value for the array element

Description

Sets the given element of the array to the specified `Object`.

SetStatic<Type>Field

Name

`SetStatic<Type>Field` —

Synopsis

```
void SetStaticBooleanField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                            jboolean newValue )
void SetStaticByteField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                         jbyte newValue )
void SetStaticCharField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                         jchar newValue )
void SetStaticDoubleField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                           jdouble newValue )
void SetStaticFloatField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                          jfloat newValue )
void SetStaticIntField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                        jint newValue )
void SetStaticLongField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                         jlong newValue )
void SetStaticObjectField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                           jobject newValue )
void SetStaticShortField( JNIEnv *env, jclass clazz, jfieldID fieldID,
                          jshort newValue )
```

Parameters

`env`

The native interface pointer

`clazz`

The class in which to set the field value

`fieldID`

A unique identifier for the desired field `newValue`

The value to which the field should be set

Description

`SetStaticBooleanField()`, `SetStaticByteField()`, `SetStaticCharField()`, `SetStaticDoubleField()`, `SetStaticFloatField()`, `SetStaticIntField()`, `SetStaticLongField()`, `SetStaticObjectField()`, and `SetStaticShortField()` each sets the value of a class (`static`) field of the appropriate Java type to a new value within the given class.

Throw

Name

`Throw` —

Synopsis

```
jint Throw( JNIEnv *env, jthrowable obj )
```

Parameters

`env`

The native interface pointer

`obj`

A `Throwable` object

Description

Throws a precreated `Throwable` object, typically an `Exception` of some type.

ThrowNew

Name

`ThrowNew` —

Synopsis

```
jint ThrowNew( JNIEnv *env, jclass clazz, const char *message )
```

Parameters

`env`

The native interface pointer

`clazz`

The class information for the object to be thrown; typically `Exception` or `Error` object of type `Throwable`

`message`

The text message for the `Throwable` object

Description

Creates a new `Throwable` object from the given class and throws it.

ToReflectedField

Name

`ToReflectedField` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`clazz`

The class that contains the field

`fieldID`

A unique identifier for the desired field

`isStatic`

Whether the field ID specifies a class (`static`) field

Description

Converts the specified field ID into an instance of the `java.lang.reflect.Field` class.

ToReflectedMethod

Name

`ToReflectedMethod` —

Synopsis

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

Availability

Java 2 SDK Version 1.2 and later

Parameters

`env`

The native interface pointer

`clazz`

The class that contains the method

`fieldID`

A unique identifier for the desired method

`isStatic`

Whether the method ID specifies a class (`static`) method

Description

Converts the specified method ID into an instance of the `java.lang.reflect.Method` class.

UnregisterNatives

Name

`UnregisterNatives` —

Synopsis

```
jint UnregisterNatives( JNIEnv *env, jclass clazz )
```

Parameters

`env`

The native interface pointer

`clazz`

The class with which the methods are associated

Description

Unregisters native methods associated with a class.

Chapter 16. RNI Function Reference

Array Handling

ArrayAlloc

Name

`ArrayAlloc` —

Synopsis

```
        HObject *ArrayAlloc( int type, int cItems )
```

Parameters

`type`

The type of array to allocate. Valid values include T_FLOAT, T_DOUBLE, T_BYTE, T_SHORT, T_INT, T_LONG and T_CHAR

`cItems`

The number of elements in the array

Description

ArrayAlloc() is used to create arrays of primitive datatypes, for example arrays of floating point values or integers.

ArrayCopy

Name

`ArrayCopy` —

Synopsis

```
void ArrayCopy( HObject *srch, long src_pos,
                HObject *dsth,
                long dst_pos,
                long length )
```

Parameters

`srch`

The source array to copy data from

`src_pos`

The position of the first element to copy from the source array

`dsth`

The destination array to copy data to

`dst_pos`

The array element to copy the data to

`length`

The number of elements to copy

Description

This function acts in a similar way to System.arraycopy() in that elements from one array are transferred to the other.

ClassArrayAlloc

Name

`ClassArrayAlloc` —

Synopsis

```
        HObject *ClassArrayAlloc( int type, int cItems,
                                   char *szSig )
```

Parameters

`type`

The type of array to allocate. All the primitive datatype values are valid here as is T_CLASS signifying an array of Java Objects

`cItems`

The number of elements in the array

`szSig`

The signature of the class to create an array of if the type parameter is T_CLASS

Description

ClassArrayAlloc() creates a new Java array of the given length of either primitive datatypes or Java Objects depending on the type parameter.

ClassArrayAlloc2

Name

`ClassArrayAlloc2` —

Synopsis

```
HObject * ClassArrayAlloc2( int type, int cItems,
                            ClassClass * cb )
```

Parameters

`type`

The type of array to allocate. All the primitive data type values are valid here as is T_CLASS signifying an array of Java Objects

`cItems`

The number of elements in the array to allocate

`cb`

The address of the class object to be used to allocate the array elements

Description

Allocates a Java array of primitive data types or Java Objects

Class Handling

AddPathClassSource

Name

`AddPathClassSource` —

Synopsis

```
        BOOL AddPathClassSource( const char * path,
                                 BOOL fAppend )
```

Parameters

`path`

The path that is to be appended to prepended to the class path

`fAppend`

If true, the given path is appended to the class path, otherwise it is prepended

Description

Dynamically adds a path to the Microsoft VM’s internal class path

ClassClassToClassObject

Name

`ClassClassToClassObject` —

Synopsis

```
HObject * ClassClassToClassObject( ClassClass * cls )
```

Parameters

`cls`

The address of the class object

Description

Retrieves a java.lang.Class object for a given ClassClass value.

ClassObjectToClassClass

Name

`ClassObjectToClassClass` —

Synopsis

```
ClassClass * ClassObjectToClassClass(
                                      HObject * object )
```

Parameters

`object`

A java.lang.Class object

Description

Retrives the ClassClass pointer for a given Java class

Class_GetAttributes

Name

`Class_GetAttributes` —

Synopsis

```
int Class_GetAttributes( ClassClass *cls )
```

Parameters

`cls`

The class to fetch the attributes from

Description

This function returns a bitmask of the ACC_*constants that are applicable to the given class

Class_GetField

Name

`Class_GetField` —

Synopsis

```
        struct fieldblock *Class_GetField(
                                            ClassClass *cls,
                                            const char *name )
```

Parameters

`cls`

The classblock of the pertinent class

`name`

The name of the field to return the fieldblock information for

Description

Class_GetField() returns the fieldblock information for the desired field in the given class specified by the classblock

Class_GetFieldByIndex

Name

`Class_GetFieldByIndex` —

Synopsis

```
struct fieldblock *Class_GetFieldByIndex(
                                           ClassClass *cls,
                                           unsigned index )
```

Parameters

`cls`

The classblock of the pertinent class

`index`

The index of the field to return the fieldblock information for

Description

This function returns the fieldblock information for the field given by the field index

Class_GetFieldCount

Name

`Class_GetFieldCount` —

Synopsis

```
unsigned Class_GetFieldCount( ClassClass *cls )
```

Parameters

`cls`

The classblock of the pertinent class

Description

This function returns the number of fields in the class including supers

Class_GetInterface

Name

`Class_GetInterface` —

Synopsis

```
ClassClass *Class_GetInterface( ClassClass *cls,
                                 unsigned index )
```

Parameters

`cls`

The class to fetch the interface from

`index`

The index number of the interface to return

Description

Class_GetInterface() returns the classblock interface for the interface located at the given index in the class

Class_GetInterfaceCount

Name

`Class_GetInterfaceCount` —

Synopsis

```
unsigned Class_GetInterfaceCount( ClassClass *cls )
```

Parameters

`cls`

The class to return the number of interfaces for

Description

Class_GetInterfaceCount() returns the number of interfaces that this class implements

Class_GetMethod

Name

`Class_GetMethod` —

Synopsis

```
        struct methodblock *Class_GetMethod(
                                            ClassClass *cls,
                                            const char *name,
                                            const char *signature )
```

Parameters

`cls`

The classblock of the class to query

`name`

The name of the method in question

`signature`

The type signature of the desired method

Description

This function returns the methodblock information for the desired method specified by the given name and signature

Class_GetMethodByIndex

Name

`Class_GetMethodByIndex` —

Synopsis

```
        struct methodblock *Class_GetMethodByIndex(
                                                   ClassClass *cls,
                                          unsigned index )
```

Parameters

`cls`

The classblock of the class to query

`index`

The index of the method to return the methodblock information for

Description

Class_GetMethodByIndex() returns the methodblock information for the method numbered by the given index in the class

Class_GetMethodCount

Name

`Class_GetMethodCount` —

Synopsis

```
unsigned Class_GetMethodCount( ClassClass *cls )
```

Parameters

`cls`

The classblock of the class to query

Description

This function returns the number of methods within the given class

Class_GetName

Name

`Class_GetName` —

Synopsis

```
        const char *Class_GetName( ClassClass *cls )
```

Parameters

`cls`

The class of which the name will be returned

Description

This function returns the fully qualified class name for the given class

Class_GetSuper

Name

`Class_GetSuper` —

Synopsis

```
        ClassClass *Class_GetSuper( ClassClass *cls )
```

Parameters

`cls`

The class of which to return the superclass of

Description

Class_GetSuper() returns the classblock information for the superclass of the given class

FindClass

Name

`FindClass` —

Synopsis

```
        ClassClass *FindClass( ExecEnv *ee,
                               char *classname,
                               bool_t resolve )
```

Parameters

`ee`

The execution environment. This should be passed a value of NULL.

`classname`

The fully qualified name of the class to locate. The package separator character is a slash ’/’

`resolve`

This flag specifies whether or not the class should be resolved. In the Microsoft JVM, this flag is ignored.

Description

FindClass() locates the classblock information for the given class which can be used in other methods, such as execute_java_constructor().

FindClassEx

Name

`FindClassEx` —

Synopsis

```
        ClassClass *FindClassEx( char *pszClassName,
                                  DWORD dwFlags )
```

Parameters

`pszClassName`

The fully qualified name of the class to locate. The package separator character is a slash ’/’

`dwFlags`

This flag specifies the mode in which this function should operate. A value of FINDCLASSEX_NOINIT prevents the static initializers of the class from executing. A value of

FINDCLASSEX_IGNORECASE performs an case-insensitive search for the class name and a value of FINDCLASSEX_SYSTEMONLY searchs for the named class only if a system class.

Description

FindClass() locates the classblock information for the given class which can be used in other methods, such as execute_java_constructor().

FindClassFromClass

Name

`FindClassFromClass` —

Synopsis

```
ClassClass * FindClassFromClass(
                                    const char * pszClassName,
                                    DWORD dwFlags,
                                    ClassClass * pContextClass )
```

Parameters

`pszClassName`

The name of the class to locate

`dwFlags`

This flag specifies the mode in which this function should operate. A value of FFINDCLASSEX_NOINIT prevents the static initializers of the class from executing. A value of FINDCLASSEX_IGNORECASE performs an case-insensitive search for the class name and a value of FINDCLASSEX_SYSTEMONLY searches for for the named class only if a system class `pContextClass`

The ClassLoader class context

Description

FindClass() locates the classblock information for the given class which can be used in other methods, such as execute_java_constructor(). This method is similar to FindClassEx() but takes an extra ClassClass * parameter specifying the ClassLoader context to use

Debugging

jio_snprintf

Name

`jio_snprintf` —

Synopsis

```
int jio_snprintf( char *str, size_t count,
                  const char *fmt,
                  ... args )
```

Parameters

`str`

The string to print to

`count`

The maximum number of characters to print `fmt`

The format string as used by printf()

`args`

Any other arguments specified in the format

Description

Prints to a string with a capped number of characters

jio_vsnprintf

Name

`jio_vsnprintf` —

Synopsis

```
int jio_vsnprintf( char *str, size_t count,
                   const char *fmt,
                   va_list args )
```

Parameters

`str`

The string to print to

`count`

The maximum number of characters to print

`fmt`

The format string as used by printf()

`args`

The arguments specified in the format string

Description

Prints to a string with argument values stored as a variable-length list

Exception Handling

HResultFromException

Name

`HResultFromException` —

Synopsis

```
HResult HResultFromException(
                              OBJECT *exception_object )
```

Parameters

`exception_object`

The Exception object to convert

Description

HResultFromException() converts a Java Exception object into an HRESULT object

SignalError

Name

`SignalError` —

Synopsis

```
        void SignalError( struct execenv *ee, char *ename,
                          char *detailMessage )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL.

`ename`

The fully qualified class name of the Exception you wish to be thrown

`detailMessage`

An explanatory message associated with the Exception

Description

This function creates a new object of type Throwable from the specified class, attaches the specified method and throws the exception

SignalErrorHResult

Name

`SignalErrorHResult` —

Synopsis

```
        void SignalErrorHResult( HRESULT theHRESULT )
```

Parameters

`theHRESULT`

    The standard return value signifying success or failure

Description

Creates a Java Exception object from an HRESULT object.

SignalErrorPrintf

Name

`SignalErrorPrintf` —

Synopsis

```
        void SignalErrorPrintf( char *ename,
                                char *pszFormat,
                                ... args )
```

Parameters

`ename`

The fully qualified class name of the Exception object you wish thrown

`pszFormat`

A format string in the form used by printf

`args`

Any other arguments specified in the format string

Description

This function creates a new object from the given class and attaches a message specified by the format string and arguments. The new exception object is then thrown

exceptionClear

Name

`exceptionClear` —

Synopsis

```
void exceptionClear( ExecEnv *ee )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL

Description

exceptionClear() removes any exceptions waiting to be thrown. The program then acts as if the exceptions had never been thrown in the first place

exceptionDescribe

Name

`exceptionDescribe` —

Synopsis

```
void exceptionDescribe( ExecEnv *ee )
```

Parameters

`ee`

The execution environment. This should be passed a value a NULL

Description

This function actually invokes the printStackTrace() method of the Throwable class. This displays, usually to System.err, the backtrace of why a particular exception had been thrown

exceptionOccurred

Name

`exceptionOccurred` —

Synopsis

```
bool_t exceptionOccurred( ExecEnv *ee )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL

Description

exceptionOccurred() tests to see whether or not an exception has been signalled at any point so far during a native method's execution

exceptionSet

Name

`exceptionSet` —

Synopsis

```
void exceptionSet( ExecEnv *ee,
                   HObject *phThrowable )
```

Parameters

`ee`

The execution environment.

`phThrowable`

A pointer to the Exception object that is to be set

Description

Sets the pending Exception to be thrown to be the given Exception object

getPendingException

Name

`getPendingException` —

Synopsis

```
HObject *getPendingException( ExecEnv *ee )
```

Parameters

`ee`

The execution environment. The value of NULL should always be passed for this argument.

Description

This function returns the current pending Exception. If no exceptions are pending, NULL is returned.

Field Handling

Field_Get<Type>

Name

`Field_Get<Type>` —

Synopsis

```
        <rtype> Field_Get<Type>( HObject *obj,
                                     struct fieldblock *field )
```

Parameters

`obj`

The object to fetch the field value from

`field`

The fieldblock information for the appropriate field

Description

This function returns the value stored in the given field of the given object. The available RNI functions and return types are as follows:

```
    RNI Return Type     Function
    --------------------------------
    bool_t              Field_GetBoolean
    signed char         Field_GetByte
    unicode             Field_GetChar
    short               Field_GetShort
    __int32             Field_GetInt
    __int64             Field_GetLong
```

`float`	`Field_GetFloat`
`double`	`Field_GetDouble`
`HObject *`	`Field_GetObject`

Field_GetOffset

Name

`Field_GetOffset` —

Synopsis

```
unsigned Field_GetOffset( struct fieldblock *field )
```

Parameters

`field`

The address of fieldblock

Description

Returns the offset of dynamic fields in the class

Field_GetStaticPtr

Name

`Field_GetStaticPtr` —

Synopsis

```
PVOID Field_GetStaticPtr( struct fieldblock *field )
```

Parameters

`field`

The address of a fieldblock

Description

Returns the address of static data in the class

Field_GetValue

Name

`Field_GetValue` —

Synopsis

```
__int32 Field_GetValue( HObject *obj,
                        struct fieldblock *field )
```

Parameters

`obj`

The object to fetch the field value from `field`

The fieldblock information for the appropriate field

Description

This function returns the value stored in the given field in the given object where the field is of datatype int, byte, char or short

Field_GetValue64

Name

`Field_GetValue64` —

Synopsis

```
        __int64 Field_GetValue64( HObject *obj,
                                  struct fieldblock *field )
```

Parameters

`obj`

The object to fetch the field value from

`field`

The fieldblock information for the appropriate field

Description

This function returns the value stored in the given field in the given object where the field is of datatype long

Field_Set<Type>

Name

`Field_Set<Type>` —

Synopsis

```
        void Field_Set<Type>( HObject *obj,
                              struct fieldblock *field,
                              <rtype> value )
```

Parameters

`obj`

The object to set the field value in

`field`

The fieldblock information for the appropriate field

`value`

The new value for the field

Description

This function sets the value of the given field in the given object. The available RNI functions and data types are as follows:

RNI DataType	Function
bool_t	Field_SetBoolean
signed char	Field_SetByte
unicode	Field_SetChar
short	Field_SetShort
__int32	Field_SetInt

`__int64`	`Field_SetLong`
`float`	`Field_SetFloat`
`double`	`Field_SetDouble`
`HObject *`	`Field_SetObject`

Field_SetValue

Name

`Field_SetValue` —

Synopsis

```
        void Field_SetValue( HObject *obj,
                             struct fieldblock *field,
                             __int32 value )
```

Parameters

`obj`

The object to set the field value in

`field`

The fieldblock information for the appropriate field

`value`

The new value for the function

Description

This function sets the value of the given field in the given object where the field is of datatype int, byte, char or short

Field_SetValue64

Name

`Field_SetValue64` —

Synopsis

```
        void Field_SetValue64( HObject *obj,
                               struct fieldblock *field,
                               __int64 value )
```

Parameters

`obj`

The object to set the field value in

`field`

The fieldblock information for the appropriate field

`value`

The new value for the field

Description

This function sets the value of the given field in the given object where the field is of datatype long

Garbage Collection

GCDisable

Name

`GCDisable` —

Synopsis

```
        int GCDisable( )
```

Description

Disable garbage collection

GCDisableCount

Name

`GCDisableCount` —

Synopsis

```
        int GCDisableCount( )
```

Description

Disables the counting performed by the garbage collector for reference usage

GCDisableMultiple

Name

`GCDisableMultiple` —

Synopsis

```
void GCDisableMultiple( int cDisable )
```

Parameters

`cDisable`

The number of times to increment the block count

Description

Increments the block count a given number of times

GCEnable

Name

`GCEnable` —

Synopsis

```
int GCEnable( )
```

Description

Enable garbage collection

GCEnableCompletely

Name

`GCEnableCompletely` —

Synopsis

```
        int GCEnableCompletely( )
```

Description

Resets the reference count to 0 and enables garbage collection immediately

GCFramePop

Name

`GCFramePop` —

Synopsis

```
        void GCFramePop( PVOID pGCFrame )
```

Parameters

`pGCFrame`

The GCFrame to pop off the protected stack

Description

This function removes any garbage collector tracking of the objects stored within the GCFrame

GCFramePush

Name

`GCFramePush` —

Synopsis

```
void GCFramePush( PVOID pGCFrame, PVOID pObjects,
                  DWORD cbObjectStructSize )
```

Parameters

`pGCFrame`

The GCFrame to push onto the protected stack

`pObjects`

The objects to protect

`cbObjectStructSize`

The size of the GCFrame that you wish to protect

Description

This function specifies object references that should be automatically tracked by the garbage collector

GCFreeHandle

Name

`GCFreeHandle` —

Synopsis

```
        void GCFreeHandle( HObject **pphobj )
```

Parameters

`pphobj`

The strong pointer to free

Description

GCFreeHandle() releases a strong pointer to an object

GCFreePtr

Name

`GCFreePtr` —

Synopsis

```
void GCFreePtr( HObject **pphobj )
```

Parameters

`pphobj`
: The previously allocate weak pointer to release

Description

GCFreePtr() releases a weak pointer allowing the object to be garbage collected as normal

GCGetPtr

Name

`GCGetPtr` —

Synopsis

```
HObject **GCGetPtr( HObject *phobj )
```

Parameters

`phobj`
: The object to return a weak pointer for

Description

This function returns a new weak pointer for the given object which disallows the garbage collector from moving the object about

GCNewHandle

Name

`GCNewHandle` —

Synopsis

```
HObject **GCNewHandle( HObject *phobj )
```

Parameters

`phobj`

The object to allocate a strong pointer for

Description

This function allocates a strong pointer to the object

GCSetObjectReferenceForHandle

Name

`GCSetObjectReferenceForHandle` —

Synopsis

```
void GCSetObjectReferenceForHandle(
                                    HObject **handle,
                                    HObject *phobj )
```

Parameters

`handle`

The handle to update

`phobj`

The new handle value

Description

GCSetObjectReferenceForHandle() allows the altering of the value of a handle in a GC-safe way. Attempts to alter a handle without using this method will generally cause the JVM to crash

GCSetObjectReferenceForObject

Name

`GCSetObjectReferenceForObject` —

Synopsis

```
void GCSetObjectReferenceForObject(
                                    HObject **location,
                                    HObject *object )
```

Parameters

`location`

Pointer to a field within an object

`object`

The new object pointer

Description

GCSetObjectReferenceForObject() allows the updating of a field value within in an object in a GC-safe way.

JVM Embedding

PrepareThreadForJava

Name

`PrepareThreadForJava` —

Synopsis

```
BOOL PrepareThreadForJava( PVOID pThreadEntryFrame )
```

Parameters

`pThreadEntryFrame`

An instance of a thread entry frame that will be populated upon a succesful attach to a JVM

Description

This function should be called within standalong native programs to enable access to the JVM from those programs. No RNI calls should be made until after PrepareThreadForJava() has successfully executed.

PrepareThreadForJavaEx

Name

`PrepareThreadForJavaEx` —

Synopsis

```
        BOOL PrepareThreadForJavaEx( PVOID pThreadEntryFrame,
                                     DWORD flags )
```

Parameters

`pThreadEntryFrame`

An instance of a thread entry frame that will be populated upon a succesful attach to a JVM

`flags`

Flags specifying extended entry behaviour. For example, if the PTJF_DONTINSTALLSTANDARDSECURITY bit is defined, the default security manager is not installed for the first caller that initializes the VM

Description

This function is an extension of PrepareThreadForJava() and should be called within standalong native programs to enable access to the JVM from those programs. No RNI calls should be made until after PrepareThreadForJava() has successfully executed.

UnprepareThreadForJava

Name

`UnprepareThreadForJava` —

Synopsis

```
        VOID UnprepareThreadForJava(
                                             PVOID pThreadEntryFrame )
```

Parameters

`pThreadEntryFrame`

An instance of a thread entry frame. This should point at the same frame used in PrepareThreadForJava() when entering the JVM

Description

UnprepareThreadForJava() disassociates a given OS thread from an instance of a JVM. Any RNI calls made within this thread after UnprepareThreadForJava() is called will cause JVM instability or simply crash.

Member Information

Member_GetAttributes

Name

`Member_GetAttributes` —

Synopsis

```
int Member_GetAttributes( PVOID member )
```

Parameters

`member`

The address of a methodblock or fieldblock

Description

Returns the attributes of the class the method or field is implemented in

Member_GetClass

Name

`Member_GetClass` —

Synopsis

```
ClassClass *Member_GetClass( PVOID member )
```

Parameters

`member`

The address of a methodblock or fieldblock

Description

Returns the name of the class the method or field belong to

Member_GetName

Name

`Member_GetName` —

Synopsis

```
const char *Member_GetName( PVOID member )
```

Parameters

`member`

The address of a methodblock or fieldblock

Description

Returns the name of a method or field

Member_GetSignature

Name

`Member_GetSignature` —

Synopsis

```
        const char *Member_GetSignature( PVOID member )
```

Parameters

`member`

The address of a methodblock or fieldblock

Description

Returns the signature of a method or field

Method Handling

do_execute_java_method

Name

`do_execute_java_method` —

Synopsis

```
long do_execute_java_method( ExecEnv *ee,
                             void *obj,
                             char *method_name,
                             char *signature,
                             struct methodblock *mb,
                             bool_t isStaticCall,
                             ... args )

long do_execute_java_methodV( ExecEnv *ee,
                              void *obj,
                              char *method_name,
                              char *signature,
                              struct methodblock *mb,
                              bool_t isStaticCall,
                              va_list args )
```

Parameters

`ee`

The execution environment. This should be the value NULL.

`obj`

The object or class against which the method should be invoked `method_name`

  The name of the method to invoke

`signature`

  The type signature of the method to invoke

`mb`

  A previously located methodblock for the given method

`isStaticCall`

  Specifies whether or not the method to invoke has been declared as being static or not

`args`

  Any other arguments required by the Java method

Description

This function invokes a method specified by the given name and type signature. In order to provide high-performance when invoking Java methods, this method also uses pre-fetched methodblock information instead of fetching the method block for each invocation of the method. The arguments for the Java method should be supplied as a comma-separated list of values except when `do_execute_java_methodV()` is used when an ANSI varargs list should be used.

execute_java_constructor, execute_java_constructorV

Name

`execute_java_constructor, execute_java_constructorV` —

Synopsis

```
HObject *execute_java_constructor( ExecEnv *ee,
                                   char *classname,
                                   ClassClass *cb,
                                   char *signature,
                                   ... args )

HObject *execute_java_constructorV( ExecEnv *ee,
                                   char *classname
                                   ClassClass *cb,
                                   char *signature
                                   va_list args )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL.

`classname`

The fully qualified name of the class from which an object should be instantiated from

`cb`

The classblock describing the class to instantiate the object from

`signature`

The type signature of the constructor to invoke

`args`

Any other arguments required by the constructor

Description

execute_java_constructor() executes one of the constructors of the given class and instantiates a new object of that class. The new object is returned. Any arguments to the constructor are specified as a comma-separated list.

execute_java_constructorV() operates in a similar manner but the arguments to be supplied to the constructor are specified as an ANSI varargs list.

execute_java_constructor_method

Name

`execute_java_constructor_method` —

Synopsis

```
void execute_java_constructor_method( struct methodblock * mb,
                                      ... args )

void execute_java_constructor_methodV( struct methodblock *mb,
                                       va_list args )
```

Parameters

`mb`

A pointer to the methodblock structure for the desired constructor

`args`

Arguments

Description

Invokes the given Java constructor method with the arguments for the constructor being given as a comma-separated list of values. Invoking `execute_java_constructor_methodV()` will produce the same result, but the arguments should be specified as an ANSI varargs list instead of a comma-separated list.

execute_java_dynamic_method, execute_java_dynamic_method64,

execute_java_dynamic_methodV

Name

`execute_java_dynamic_method`, `execute_java_dynamic_method64`, `execute_java_dynamic_methodV` —

Synopsis

```
long execute_java_dynamic_method( ExecEnv *ee,
                                  HObject *obj,
                                  char *methodname,
                                  char *signature,
                                  ... args )

int64_t execute_java_dynamic_method64( ExecEnv *ee,
                                       HObject *obj,
                                       char *methodname,
                                       char *signature,
                                       ... args )

int64_t execute_java_dynamic_methodV( ExecEnv *ee,
                                      HObject *obj,
                                      char *methodname,
                                      char *signature,
                                      va_list args )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL.

`obj`

The object against which to invoke the method

`methodname`

The name of the method to invoke

`signature`

The type signature of the method to invoke

`args`

Any other arguments required by the method

Description

execute_java_dynamic_method() invokes an instance method in the given object. The method name and type signature should match up to provide a valid method to invoke. Any arguments to the method are specified as a comma-separated list.

execute_java_dynamic_method64() operates in an identical way to execute_java_dynamic_method(), but returns a 64-bit value from the Java method as opposed to, typically, a long int.

execute_java_dynamic_methodV() operates as per execute_java_dynamic_method(), but any arguments to the method are specified in an ANSI varargs list instead of a comma-separated list.

execute_java_interface_method, execute_java_interface_method64, execute_java_interface_methodV

Name

`execute_java_interface_method, execute_java_interface_method64, execute_java_interface_methodV` —

Synopsis

```
long execute_java_interface_method( ExecEnv *ee,
                                    HObject *pobj,
                                    ClassClass j_interface,
                                    char *methodname,
                                    char *signature,
                                    ... args )

int64_t execute_java_interface_method64( ExecEnv *ee,
                                         HObject *pobj,
                                         ClassClass j_interface,
                                         char *methodname,
                                         char *signature,
                                         ... args )

int64_t execute_java_interface_methodV( ExecEnv *ee,
                                        HObject *pobj,
                                        ClassClass j_interface,
                                        char *methodname,
                                        char *signature,
                                        va_list args )
```

Parameters

`ee`

The execution environment. This should be given a value of NULL.

`pobj`

The object against which to invoke the method

`j_interface`

The Java interface to invoke the method in

`methodname`

The name of the method to invoke

`signature`

The type signature of the method to invoke

`args`

Any other arguments required by the method

Description

execute_java_interface_method() invokes a method against the the given object defined within a given interface. The method name and type signature should match up to provide a valid method to invoke. Any arguments required by the Java method are specified as a comma-separated list.

execute_java_interface_method64() operates in an identical way to execute_java_interface_method(), but returns a 64-bit value instead of a long int.

execute_java_interface_methodV() is identical in operation to execute_java_interface_method() other than that any arguments to be passed to the Java method are supplied in an ANSI varargs list.

execute_java_static_method, execute_java_static_method64, execute_java_static_methodV

Name

`execute_java_static_method, execute_java_static_method64, execute_java_static_methodV` —

Synopsis

```
long execute_java_static_method( ExecEnv *ee,
                                 ClassClass *cb,
                                 char *method_name,
                                 char *signature,
                                 ... args )

int64_t execute_java_static_method64( ExecEnv *ee,
                                      ClassClass *cb,
                                      char *method_name,
                                      char *signature,
                                      ... args )

int64_t execute_java_static_methodV( ExecEnv *ee,
                                     ClassClass *cb,
                                     char *method_name,
                                     char *signature,
                                     ... args )
```

Parameters

`ee`

The execution environment. This should be passed a value of NULL.

`cb`

A classblock value describing the class in which the static method to invoke is defined

`method_name`

The name of the method to invoke

`signature`

The type signature of the method to invoke

`args`

Any other arguments required by the specified method

Description

execute_java_static_method() invokes a method declared within a Java class as being static, i.e., class-level and not instance-level. Any arguments required by the Java method are specified as a comma-separated list.

execute_java_static_method64() has the same operation but returns a 64-bit value. execute_java_static_methodV() also provides the same functionality but any arguments required by the Java method are supplied in the form of an ANSI varargs list.

get_methodblock

Name

`get_methodblock` —

Synopsis

```
        struct methodblock *get_methodblock(
                                            HObject *pjobj,
                                            char *methodname,
                                            char *signature )
```

Parameters

`pjobj`

The object containing the method which you wish to retrieve the methodblock for

`methodname`

The name of the method to return the methodblock for

`signature`

The type signature for the desired method

Description

get_methodblock() returns information on a given method within a class which can be used in conjunction with do_execute_java_method(), a more optimized function for invoking Java methods from native code.

Miscellaneous

AddModuleResourceClassSource

Name

`AddModuleResourceClassSource` —

Synopsis

```
void AddModuleResourceClassSource( HMODULE hMod,
                                   DWORD dwResID )
```

Parameters

`hMod`

A handle to a module

`dwResID`

The unique resource ID

Description

Notifies the Microsoft JVM of a Win32 resource containing class files. When classes are loaded, the resource is searched for the appropriate classes as if it were a directory on the class path

GetCurrentJavaTimeMillis

Name

`GetCurrentJavaTimeMillis` —

Synopsis

```
__int64 GetCurrentJavaTimeMillis( )
```

Description

Returns the same result as defined by java.lang.System.currentTimeMillis()

GetNativeMethodCallersClass

Name

`GetNativeMethodCallersClass` —

Synopsis

```
ClassClass *GetNativeMethodCallersClass( )
```

Description

Returns the class information for the class or object whose method was invoked to enter the current native method

GetNativeMethodCallersMethodInfo

Name

`GetNativeMethodCallersMethodInfo` —

Synopsis

```
struct methodblock *GetNativeMethodCallersMethodInfo( )
```

Description

Returns the method information about the natively declared method invoked to enter the current native method

GetNativeMethodsClass

Name

`GetNativeMethodsClass` —

Synopsis

```
ClassClass * GetNativeMethodsClass( )
```

Description

Retrieves information about the currently executing native method’s class

GetNativeMethodsMethodInfo

Name

`GetNativeMethodsMethodInfo` —

Synopsis

```
struct methodblock * GetNativeMethodsMethodInfo( )
```

Description

Retrieves information about the currently executing native method

RNIGetCompatibleVersion

Name

`RNIGetCompatibleVersion` —

Synopsis

```
DWORD RNIGetCompatibleVersion( )
```

Description

This function must be defined in your native libraries returning the value of RNIVER as defined with <native.h> in order to load into the JVM.

Thread_IsInterrupted

Name

`Thread_IsInterrupted` —

Synopsis

```
        BOOL Thread_IsInterrupted( BOOL fResetInterruptedFlag )
```

Parameters

`fResetInterruptedFlag`

Indicates whether or not to reset the interrupt flag

Description

Checks to see if the current thread has been interrupted and resets the interrupt flag, if desired. This is used within RNI code to determine whether or not Thread.interrupt() has been called from within another thread and wants the current thread to stop

Monitors

ObjectMonitorEnter

Name

`ObjectMonitorEnter` —

Synopsis

```
void ObjectMonitorEnter( HObject *object )
```

Parameters

`object`

A synchronization object

Description

ObjectMonitorEnter() is actually a macro which simply takes the given Java object, casts it to an unsigned int and invokes monitorEnter() for you.

ObjectMonitorExit

Name

`ObjectMonitorExit` —

Synopsis

```
void ObjectMonitorExit( HObject *object )
```

Parameters

`object`

A synchronization object

Description

ObjectMonitorExit() is actually a macro which simply takes the given Java object, casts it to an unsigned int and invokes monitorExit() for you.

ObjectMonitorNotify

Name

`ObjectMonitorNotify` —

Synopsis

```
void ObjectMonitorNotify( HObject *object )
```

Parameters

`object`

A synchronization object

Description

ObjectMonitorNotify() is actually a macro which simply takes the given Java object, casts it to an unsigned int and invokes monitorNotify() for you.

ObjectMonitorNotifyAll

Name

`ObjectMonitorNotifyAll` —

Synopsis

```
void ObjectMonitorNotifyAll( HObject *object )
```

Parameters

`object`

A synchronization object

Description

ObjectMonitorNotifyAll() is actually a macro which simply takes the given Java object, casts it to an unsigned int and invokes monitorNotifyAll() for you.

ObjectMonitorWait

Name

`ObjectMonitorWait` —

Synopsis

```
void ObjectMonitorWait( HObject *object )
```

Parameters

`object`

A synchronization object

Description

ObjectMonitorWait() is actually a macro which simply takes the given Java object, casts it to an unsigned int and invokes monitorWait() for you.

monitorEnter

Name

`monitorEnter` —

Synopsis

```
void monitorEnter( unsigned int object )
```

Parameters

`object`

A synchronization object

Description

monitorEnter() causes synchronization to occur upon the given object. This is equivalent to using the synchronized keyword within Java code

monitorExit

Name

`monitorExit` —

Synopsis

```
void monitorExit( unsigned int object )
```

Parameters

`object`
: A synchronization object

Description

This function releases the synchronization lock held on the given object

monitorNotify

Name

`monitorNotify` —

Synopsis

```
void monitorNotify( unsigned int object )
```

Parameters

`object`

A synchronization object

Description

This function wakes up a single thread waiting on the given synchronization object. This is equivalent to the java.lang.Object.notify() method in Java code.

monitorNotifyAll

Name

`monitorNotifyAll` —

Synopsis

```
        void monitorNotifyAll( unsigned int object )
```

Parameters

`object`

A synchronization object

Description

This function wakes up all threads waiting on the given synchronization object. This is equivalent to the java.lang.Object.notifyAll() method in Java code.

monitorWait

Name

`monitorWait` —

Synopsis

```
void monitorWait( unsigned int object )
```

Parameters

`object`

A synchronization object

Description

This function causes the monitor to wait until the synchronization object is notified by another thread. This is equivalent to the java.lang.Object.wait() method in Java code.

Object Characteristics

ImplementsInterface

Name

`ImplementsInterface` —

Synopsis

```
BOOL ImplementsInterface( ClassClass *cb,
                          ClassClass *icb,
                          ExecEnv *ee )
```

Parameters

`cb`

The class information to test

`icb`

The class information for the interface in question

`ee`

The execution environment. This should always have the value of NULL passed to it

Description

This function determines whether or not the given class implements the given interface

isInstanceOf

Name

`isInstanceOf` —

Synopsis

```
BOOL isInstanceOf( JHandle *phobj,
                   char *classname )
```

Parameters

`phobj`

Handle to a Java object

`classname`

The fully qualified classname of the class to test against with a package delimiter of slash ’/’

Description

isInstanceOf() tests an object to see if it is an instantiation of the given class.

is_instance_of

Name

`is_instance_of` —

Synopsis

```
        BOOL is_instance_of( JHandle *phobj,
                             ClassClass *dcb,
                             ExecEnv *ee )
```

Parameters

`phobj`

Handle to a Java object

`dcb`

Class information for the target class

`ee`

The execution environment. A value of NULL should be passed as this argument.

Description

is_instance_of() tests an object to see if it can be cast to the given target class.

is_subclass_of

Name

`is_subclass_of` —

Synopsis

```
        BOOL is_subclass_of( ClassClass *cb,
                             ClassClass *dcb,
                             ExecEnv *ee )
```

Parameters

`cb`

The classblock to test

`dcb`

The class information to test against

`ee`

The execution environment. A value of NULL should always be passed for this argument

Description

is_subclass_of() tests the given class to determine whether or not it is a subclass of the class specified by “dcb”

Object Information

Object_GetClass

Name

`Object_GetClass` —

Synopsis

```
ClassClass *Object_GetClass( HObject *obj )
```

Parameters

`obj`

The object to return the classblock information from

Description

This function returns the classblock information for the given object

String Handling

MakeByteString

Name

`MakeByteString` —

Synopsis

```
HArrayOfByte *MakeByteString( char *str, long len )
```

Parameters

`str`

A C string containing the data from which the Java array will be initialized from

`len`

The number of characters in the C string. The Java array will be created to this length

Description

MakeByteString() creates a new array of bytes from a C string.

javaString2CString

Name

`javaString2CString` —

Synopsis

```
char *javaString2CString( Hjava_lang_String *s,
                          char *buf,
                          int buflen )
```

Parameters

`s`

The Java String object from which the characters will be extracted

`buf`

The pre-allocated buffer to extract the characters into

`buflen`

The size of the C buffer

Description

This function extracts the characters from a Java String and copies them into a pre-allocated C buffer allowing you to manipulate Java String easily within C.

javaStringLength

Name

`javaStringLength` —

Synopsis

```
int javaStringLength( Hjava_lang_String *s )

int javaStringLengthAsCString( HString *s )
```

Parameters

`s`

The Java String to calculate the length of

Description

javaStringLength() returns the length of the given Java String. javaStringLengthAsCString() functions in a similar way but returns the length of the Java String as if it was a C string.

javaStringStart

Name

`javaStringStart` —

Synopsis

```
        unicode *javaStringStart( HString *string )
```

Parameters

`string`

The Java String

Description

javaStringStart() returns a pointer to the first character of the Java String object. Note that this pointer will become invalid as garbage collection occurs.

makeJavaString

Name

`makeJavaString` —

Synopsis

```
Hjava_lang_String *makeJavaString( char *str, int len )

Hjava_lang_String *makeJavaStringW( unicode *str, int len )

Hjava_lang_String *makeJavaStringFromUtf8( const char *str,
                                           int len )
```

Parameters

`str`

A C string containing the data from which the Java string will be initialized

`len`

The length of the C string data

Description

makeJavaString() and makeJavaStringFromUtf8() create a new Java String object from existing C string data in ASCII and UTF-8 format.

makeJavaStringW() similarly creates a new String object from Unicode data.