JNI

The Java Native Interface (JNI) enables the integration of code written in the Java programming language with code written in other languages such as C and C++. It allows programmers to take full advantage of the Java platform without having to abandon their investment in legacy code.

**1.1 The Java Platform and Host Environment**

Because this book covers applications written in the Java programming language

as well as in native (C, C++, etc.) programming languages, let us first clarify the

exact scope of the programming environments for these languages.

The Java platform is a programming environment consisting of the Java virtual

machine (VM) and the Java Application Programming Interface (API).1 Java

applications are written in the Java programming language, and compiled into a

machine-independent binary class format. A class can be executed on any Java

virtual machine implementation. The Java API consists of a set of predefined

classes. Any implementation of the Java platform is guaranteed to support the

Java programming language, virtual machine, and API.

The term *host environment* represents the host operating system, a set of

native libraries, and the CPU instruction set. *Native applications* are written in

*native programming languages* such as C and C++, compiled into host-specific

binary code, and linked with native libraries. Native applications and native libraries

are typically dependent on a particular host environment. A C application built

for one operating system, for example, typically does not work on other operating

systems.

Java platforms are commonly deployed on top of a host environment. For

example, the Java Runtime Environment (JRE) is a Sun product that supports the

Java platform on existing operating systems such as Solaris and Windows. The

Java platform offers a set of features that applications can rely on independent of

the underlying host environment.

**1.2 Role of the JNI**

When the Java platform is deployed on top of host environments, it may become

desirable or necessary to allow Java applications to work closely with native code

written in other languages. Programmers have begun to adopt the Java platform to

build applications that were traditionally written in C and C++. Because of the

1. As used herein, the phrases “Java virtual machine” or “Java VM” mean a virtual machine for the

Java platform. Similarly, the phrase “Java API” means the API for the Java platform.

existing investment in legacy code, however, Java applications will coexist with C

and C++ code for many years to come.

The JNI is a powerful feature that allows you to take advantage of the Java

platform, but still utilize code written in other languages. As a part of the Java virtual

machine implementation, the JNI is a *two-way* interface that allows Java

applications to invoke native code and vice versa. Figure 1.1 illustrates the role of

the JNI.



**Figure 1.1** Role of the JNI

The JNI is designed to handle situations where you need to combine Java

applications with native code. As a two-way interface, the JNI can support two

types of native code: native *libraries* and native *applications*.

**•** You can use the JNI to write *native methods* that allow Java applications to

call functions implemented in native libraries. Java applications call native

methods in the same way that they call methods implemented in the Java programming

language. Behind the scenes, however, native methods are implemented

in another language and reside in native libraries.

• The JNI supports an *invocation interface* that allows you to embed a Java virtual

machine implementation into native applications. Native applications can

link with a native library that implements the Java virtual machine, and then

use the invocation interface to execute software components written in the

Java programming language. For example, a web browser written in C can

execute downloaded applets in an embedded Java virtual machine implemention.

Java application Java virtual machine

Host environment

Native application

and library JNI and library implementation

**1.3 Implications of Using the JNI**

Remember that once an application uses the JNI, it risks losing two benefits of the

Java platform.

First, Java applications that depend on the JNI can no longer readily run on

multiple host environments. Even though the part of an application written in the

Java programming language is portable to multiple host environments, it will be

necessary to recompile the part of the application written in native programming

languages.

Second, while the Java programming language is type-safe and secure, native

languages such as C or C++ are not. As a result, you must use extra care when

writing applications using the JNI. A misbehaving native method can corrupt the

entire application. For this reason Java applications are subject to security checks

before invoking JNI features.

As a general rule, you should architect the application so that native methods

are defined in as few classes as possible. This entails a cleaner isolation between

native code and the rest of the application.

**1.4 When to Use the JNI**

Before you embark on a project using the JNI, it is worth taking a step back to

investigate whether there are alternative solutions that are more appropriate. As

mentioned in the last section, applications that use the JNI have inherent disadvantages

when compared with applications written strictly in the Java programming

language. For example, you lose the type-safety guarantee of the Java programming

language.

A number of alternative approaches also allow Java applications to interoperate

with code written in other languages. For example:

**•** A Java application may communicate with a native application through a

TCP/IP connection or through other inter-process communication (IPC)

mechanisms.

**•** A Java application may connect to a legacy database through the JDBC™

API.

• A Java application may take advantage of distributed object technologies such

as the Java IDL API.

A common characteristic of these alternative solutions is that the Java application

and native code reside in different processes (and in some cases on different

machines). Process separation offers an important benefit. The address space pro-

tection supported by processes enables a high degree of fault isolation—a crashed

native application does not immediately terminate the Java application with which

it communicates over TCP/IP.

Sometimes, however, you may find it necessary for a Java application to communicate

with native code *that resides in the same process*. This is when the JNI

becomes useful. Consider, for example, the following scenarios:

**•** The Java API might not support certain host-dependent features needed by an

application. An application may want to perform, for example, special file

operations that are not supported by the Java API, yet it is both cumbersome

and inefficient to manipulate files through another process.

**•** You may want to access an existing native library and are not willing to pay

for the overhead of copying and transmitting data across different processes.

Loading the native library in the same process is much more efficient.

**•** Having an application span multiple processes could result in unacceptable

memory footprint. This is typically true if these processes need to reside on

the same client machine. Loading a native library into the existing process

hosting the application requires less system resources than starting a new process

and loading the library into that process.

• You may want to implement a small portion of time-critical code in a lowerlevel

language, such as assembly. If a 3D-intensive application spends most of

its time in graphics rendering, you may find it necessary to write the core portion

of a graphics library in assembly code to achieve maximum performance.

In summary, use the JNI if your Java application must interoperate with native

code that resides in the same process.

**1.5 Evolution of the JNI**

The need for Java applications to interoperate with native code has been recognized

since the very early days of the Java platform. The first release of the Java

platform, Java Development Kit (JDK™) release 1.0, included a native method

interface that allowed Java applications to call functions written in other languages

such as C and C++. Many third-party applications, as well as the implementation

of the Java class libraries (including, for example, java.lang,

java.io, and java.net), relied on the native method interface to access the features

in the underlying host environment.

Unfortunately, the native method interface in JDK release 1.0 had two major

problems:

**•** First, the native code accesses fields in objects as members of C structures.

However, the Java virtual machine specification does not define how objects

are laid out in memory. If a given Java virtual machine implementation lays

out objects in a way other than that assumed by the native method interface,

then you have to recompile the native method libraries.

• Second, the native method interface in JDK release 1.0 relies on a conservative

garbage collector because native methods can get hold of direct pointers

to objects in the virtual machine. Any virtual machine implementation that

uses more advanced garbage collection algorithms cannot support the native

method interface in JDK release 1.0.

The JNI was designed to overcome these problems. It is an interface that can

be supported by all Java virtual machine implementations on a wide variety of

host environments. With the JNI:

**•** Each virtual machine implementor can support a larger body of native code.

**•** Development tool vendors do not have to deal with different kinds of native

method interfaces.

• Most importantly, application programmers are able to write one version of

their native code and this version will run on different implementations of the

Java virtual machine.

The JNI was first supported in JDK release 1.1. Internally, however, JDK

release 1.1 still uses old-style native methods (as in JDK release 1.0) to implement

the Java APIs. This is no longer the case in Java 2 SDK release 1.2 (formerly

known as JDK release 1.2). Native methods have been rewritten so that they conform

to the JNI standard.

The JNI is the native interface supported by all Java virtual machine implementations.

From JDK release 1.1 on, you should program to the JNI. The oldstyle

native method interface is still supported in Java 2 SDK release 1.2, but will

not (and cannot) be supported in advanced Java virtual machine implementations

in the future.

Java 2 SDK release 1.2 contains a number of JNI enhancements. The

enhancements are backward compatible. All future evolutions of JNI will maintain

complete binary compatibility.