Evaluating the Java Native Interface (JNI): Leveraging Existing Native Code, Libraries and Threads to a Running Java Virtual Machine

Stelios Sotiriadis, The Edward Rogers Sr. Department of Electrical and Computer Engineering, University of Toronto, Toronto, Canada
Oladotun Omosebi, Department of Computer Science, Edge Hill University, Lancashire, UK
Assem Ayapbergenova, Department of Information Technologies, Satbayev University, Almaty, Kazakhstan
Nurbek P. Saparkhojayev, Department of Information Technologies, Satbayev University, Almaty, Kazakhstan

ABSTRACT
This article aims to explore JNI features and to discover fundamental operations of the Java programming language, such as arrays, objects, classes, threads and exception handling, and to illustrate these by using various algorithms and code samples. The authors also investigate the JNI Invocation API that allows native applications to interact with the Java Virtual Machine (JVM). They focus on attaching native threads to a running JVM and on leveraging existing native code using one-to-one mapping and shared stubs.

KEYWORDS
Java Native Interface Invocation API, Java Native Interface, Java One-To-One Mapping, Java Shared Stubs, Java Virtual Machine as a Native Library, Java, Native Threads

INTRODUCTION
The Java Native Interface (JNI, 2014) aims to provide interoperability between Java code running on Java Virtual Machine (JVM) and code written in other programming languages like C, C++, assembly, python and others. It is typically used for utilizing certain platform depended features, for reusing existing libraries and for speeding up time critical execution of applications. Using the JNI, developers implement Java classes that are extended with native methods and linked dynamically. Using JNI, a Java application invokes native methods, submits parameters and obtains results after the native method has finished processing. The latter can both create and invoke operations on existing Java objects to: (a) create, update and inspect Java objects, (b) get any primitive data types or objects, (c) return either Java primitive data types or objects back to the Java program, (d) call Java instance or class methods, pass it the required parameters and get the results back when the method completes, (e) catch and throw Java exceptions and, (f) implement synchronization to support multithreaded access to a running JVMs.

Classes and Objects
JNI provides functions that allow the native code to access a field or call a method within a Java object or a Java class. Figure 1 demonstrates the classes and objects JNI methods classification.
All these functions require a value of jclass type that contains the information for the class in which the field or method is declared. JNI defines two classes namely FindClass() and GetObjectClass(), which can be used from native code to get jclass objects. Next, we outline the characteristics of each function.

1. The FindClass() method takes as parameter the fully qualified class name with package components and returns the jclass value for the given class or a NULL value. The FindClass() browses all directories and files defined in the Java classpath environment variable for a given class. The statement - jclass strClass = env -> FindClass(“Java/lang/String”) - demonstrates a call based on the FindClass() function to obtain the jclass value for the string class, where Java/lang/String is the fully qualified name for the string class. Another case involves the statement - jclass intClass = env -> FindClass(“[I”) - where the FindClass() obtains the jclass value for an array of Java integer type, with the signature being “[I”, while the JNI int[] is represented as “[I”. Finally, the FindClass() function is useful when the native code knows the fully qualified name o the class or the class signature.

2. The GetObjectClass() function takes an object as parameter and returns the jclass value of the class from which the object was instantiated, or returns a NULL value. This function is useful when the native code requires identifying the class for an object that is passed as an argument. The statement - jclass xClass = env -> GetObjectClass(x) - represents a call using the GetObjectClass() function within a native method to obtain the class of an object passed into a native method as argument.

**Accessing Fields**

A Java field represents a class, interface, or enum that has an associated value. JNI supports two types of fields; instance fields that contain unique values for every instance of the class, and static fields, which are class variables that are accessible from all the instances of a class (Liang, 1999). To access an instance field from a native code the following steps need to be performed:

1. Obtain the jclass value of the class from which the object was instantiated, by calling the GetObjectClass() function and passing in the object as a parameter. For instance, jclass cls = env -> GetObjectClass (obj).

**Figure 1. Classification of JNI Array methods**
Related Content

An Intelligent Sensor Placement Method to Reach a High Coverage in Wireless Sensor Networks
[www.igi-global.com/article/intelligent-sensor-placement-method-to-reach/56356?camid=4v1a](www.igi-global.com/article/intelligent-sensor-placement-method-to-reach/56356?camid=4v1a)

PheGee@Home: A Grid-Based Tool for Comparative Genomics
[www.igi-global.com/chapter/phegee-home-grid-based-tool/64572?camid=4v1a](www.igi-global.com/chapter/phegee-home-grid-based-tool/64572?camid=4v1a)
A Generic Reference Architecture for Collaboratory Scientific Virtual Laboratory

Adaptive Routing Strategy for Large Scale Rearrangeable Symmetric Networks