Evaluating the Java Native Interface (JNI):
Data Types and Strings

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 describes how the java native interface (JNI) is a powerful feature of the java platform that started to draw attention in the latter years as an efficient programming framework for building and delivering innovative technological applications based on disruptive technologies such as mobile, Internet of Things and embedded systems. Developers use it to incorporate native code written in programming languages such as C, C++, python etc., into java. JNI is particularly useful when java applications need to access existing native libraries or code blocks written in other languages to increase performance, avoid recoding and expand interoperability between programming languages for processes that reside in the same process. 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.

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. This is possible, for instance, by using shared objects, as in UNIX systems or Dynamic-Link Library (DLL) extensions on Windows systems. Today, with the increasing amount of applications and disruptive based innovative services like mobile applications, cloud platforms, Internet of Things (IoT) sensor protocols and development suites, the utilization of the JNI concept becomes once more essential (Sun et al., 2013). According to Dawson et al. (2015), JNI can be a key element for leveraging existing code assets in a service-oriented architecture (SOA) or cloud-based system. JNI is crucial for utilizing and interacting with libraries specific to platforms of embedded systems as it enables access to faster execution of native programs (Lee et al., 2011). This will offer a significant advantage by speeding up the development process, avoiding the reproduction of already developed code and thereby, increasing performance (Chandrian, 2011)

DOI: 10.4018/IJDST.2018040103

Copyright © 2018, IGI Global. Copying or distributing in print or electronic forms without written permission of IGI Global is prohibited.
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.

There are various ways in which a Java application can communicate with a native application. These include using a TCP/IP connection, and via Inter-Process Communications (IPC) mechanisms etc. (Stenman & Sagonas, 2009). A Java application may also connect to a legacy system via a database using standard Java API, or by utilizing distributed objects such as RPC or Java IDL API (Sperko, 2003). However, in the aforementioned cases the native code resides in different processes. Having said that, the contribution of this work focuses on exploring JNI features that are essential for cases in which a Java application will need to communicate with native code that resides in the same process (Android M Developer, 2015, Liang, 1999). This has been proposed as an efficient solution for mobile application platforms, such as the Android platform, to load code from dynamic shared libraries. In particular, from the perspective of disruptive technologies, the benefits of using JNI includes the fact that both the calling program and the called program share the same process, while still allowing other methods to start new processes. By this method JNI provides faster calls and consumes less resources.

This work presents a discussion of the vital Java features in the concept of JNI. It is organized as follows: next, we present the usage of JNI in current systems, and discuss the primitive data types, such as strings and arrays. Then, we cover classes and objects, including fields and method invocations such as instance methods, static methods, instance methods of a superclass, etc. We also discuss exceptions handling, including the catch and throw strategy and cover the JNI invocation API and the JNI threads. At the end, we talk about leveraging native code based on one-to-one mapping and shared stubs and finally, we present the conclusions.

USE OF JNI IN TODAY’S SYSTEMS

JNI is considered as a framework of the Java platform that is receiving more and more attention daily due to the emergency of new technologies such as the Internet of Things, embedded systems, security and Big Data analysis. Also, we expect that JNI will enable easy and efficient integration with technologies such as SOA and cloud computing. According to Dawson et al. (2015), JNI is a key technology for integrating non-Java legacy code and components with a Java-based platform, and thereby providing a building block for an SOA or cloud-based system. The use of JNI is vital for the following cases: (a) for already developed and considerably large and tricky code that is written in another language (e.g. C++) and that needs to be executed (i.e. code reusability), (b) for accessing system devices or to perform platform-specific tasks that cannot be executed in Java and, (c) for performance issues, due to the fast that Java can be too slow for application purposes. In this case, performance could be improved by implementing critical portions of code in other languages (e.g. C++).

Based on this, the following works demonstrate the usage of JNI in today’s applications and systems. In (Wang, 2012), the authors present the design and implementation of G-Hadoop, a MapReduce framework that aims to enable large-scale distributed data analytics. The Torque GridWay interfaces for clusters can only be implemented based on prior bindings to the native libraries. Having been inspired by bindings for the Sun Grid Engine, they implemented custom bindings to the Torque using the JNI concept. In (Sun, 2013) a security framework is presented that can extend the Java security model to control the native code.
A Grid-Aware Emergency Response Model (G-AERM) for Disaster Management
Eleana Asimakopoulou, Chimay J. Anumba and Dino Bouchlaghem (2010). 
[www.igi-global.com/article/grid-aware-emergency-response-model/46049?camid=4v1a](www.igi-global.com/article/grid-aware-emergency-response-model/46049?camid=4v1a)

Adaptive Routing Strategy for Large Scale Rearrangeable Symmetric Networks
Amitabha Chakrabarty, Martin Collier and Sourav Mukhopadhyay (2010).
*International Journal of Grid and High Performance Computing* (pp. 53-63).
[www.igi-global.com/article/adaptive-routing-strategy-large-scale/43884?camid=4v1a](www.igi-global.com/article/adaptive-routing-strategy-large-scale/43884?camid=4v1a)