Java Native Interface

Diego Rodrigo Cabral Silva

Overview

- The JNI allows Java code that runs within a Java Virtual Machine (VM) to operate with applications and libraries written in other languages, such as C, C++, and assembly.
- The *Invocation API* allows you to embed the Java Virtual Machine into your native applications.

Overview

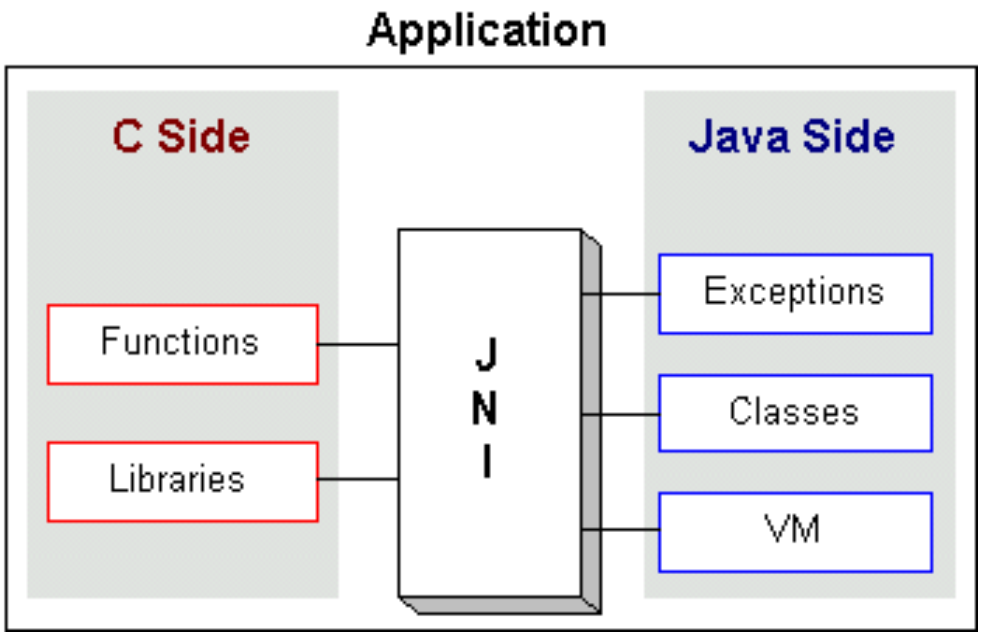
- The standard Java class library may not support the platform-dependent features needed by your application.
- You may already have a library or application written in another programming language and you wish to make it accessible to Java applications.
- You may want to implement a small portion of time-critical code in a lower-level programming language, such as assembly, and then have your Java application call these functions.

Overview

- the native language side and the Java side of an application can create, update, and access Java objects and then share these objects between them.
- The native method, using the JNI framework, can call the existing Java method, pass it the required parameters, and get the results back when the method completes.

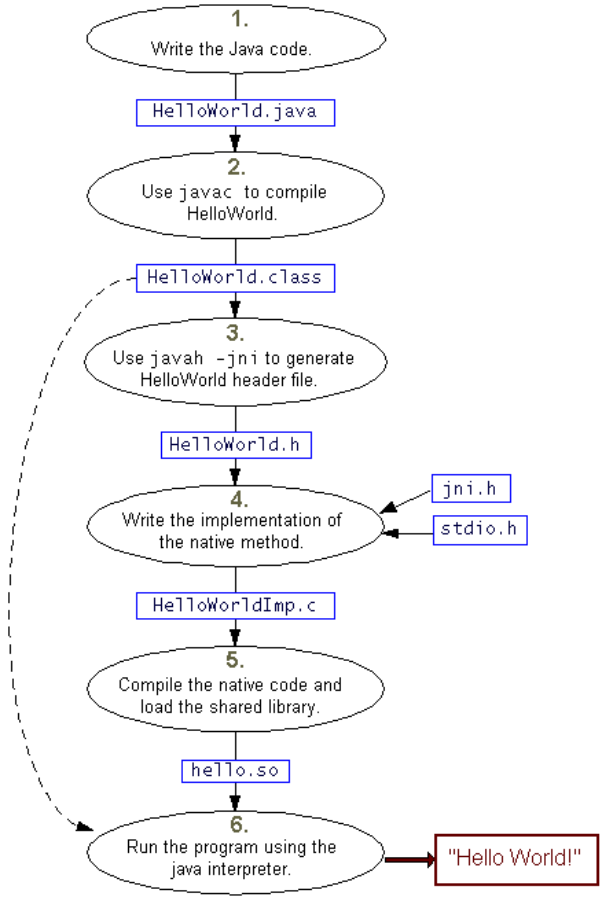
Overview

- It is easy to see that the JNI serves as the glue between Java and native applications.



Writing a Java Program with Native Methods

- Begin by writing the Java program. Create a Java class that declares the native method; this class contains the declaration or signature for the native method. It also includes a main method which calls the native method.
- Compile the Java class that declares the native method and the main method.
- Generate a header file for the native method using javah with the native interface flag -jni. Once you've generated the header file you have the formal signature for your native method.
- Write the implementation of the native method in the programming language of your choice, such as C or C++.
- Compile the header and implementation files into a shared library file.
- Run the Java program.



Write the Java Code

```
class HelloWorld {
    public native void displayHelloWorld();
    static {
        System.loadLibrary("hello");
    }
    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

Compile the Java Code

javac HelloWorld.java

Create the .h File

javah HelloWorld

Write the Native Method Implementation

```
#include <jni.h>
#include “HelloWorld.h”
#include <stdio.h>

JNIEXPORT void JNICALL
    Java_HelloWorld_displayHelloWorld(JNIEnv *env,
    jobject obj) {
        printf("Hello world!\n");
        return;
}
```

Create a Shared Library

g++ -shared -I/usr/local/java/include -I/usr/local/java/include/solaris HelloWorldImp.c -o libhello.so

Set the Library Path

```
export LD_LIBRARY_PATH=.
```

Run the Program

java HelloWorld

Mapping between Java and Native Types

Java Type	Native Type	Size in bits
`boolean`	`jboolean`	8, unsigned
`byte`	`jbyte`	8
`char`	`jchar`	16, unsigned
`short`	`jshort`	16
`int`	`jint`	32
`long`	`jlong`	64
`float`	`jfloat`	32
`double`	`jdouble`	64
`void`	`void`	n/a

Accessing Java Strings

```
JNIEXPORT jstring JNICALL
    Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring
    prompt) {
        char buf[128];
        const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
         printf("%s", str);
        (*env)->ReleaseStringUTFChars(env, prompt, str);
...
```

Accessing Arrays of primitive elements

```
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env,
jobject obj, jintArray arr) {
   int i, sum = 0;
   jsize len = (*env)->GetArrayLength(env, arr);
   jint *body = (*env)->GetIntArrayElements(env, arr, 0);
   for (i=0; i<len; i++) {
      sum += body[i];
   }
   (*env)->ReleaseIntArrayElements(env, arr, body, 0);
   return sum;
}
```