FREE RESOURCES (/S)

(/)

TRAINING & MENTORSHIPS (/PAID-TRAINING)

ABOUT (/ABOUT/INDEX.HTML)

LOGIN (/LOGIN.HTML)

E.g. Tab Navigation with JQuery, HT

Java Fundamentals Tutorial: Java ...

Java Fundamentals Tutorial: Java Native Interface (JNI)

Prev Next

16. Java Native Interface (JNI)

16.1. JNI Overview

- An interface that allows Java to interact with code written in another language
- Motivation for JNI
  - Code reusability
    - Reuse existing/legacy code with Java (mostly C/C++)
  - Performance
    - Native code used to be up to 20 times faster than Java, when running in interpreted mode
    - Modern JIT compilers (HotSpot) make this a moot point
  - Allow Java to tap into low level O/S, H/W routines
- JNI code is not portable!

Note

JNI can also be used to invoke Java code from within natively-written applications - such as those written in C/C++.

In fact, the java command-line utility is an example of one such application, that launches Java code in a Java Virtual Machine.

16.2. JNI Components

- `javah` - JDK tool that builds C-style header files from a given Java class that includes `native` methods
  - Adapts Java method signatures to native function prototypes
- `jni.h` - C/C++ header file included with the JDK that maps Java types to their native counterparts
  - `javah` automatically includes this file in the application header files

16.3. JNI Development (Java)

- Create a Java class with native method(s): `public native void sayHi(String who, int times);`
- Load the library which implements the method: `System.loadLibrary("HelloImpl");`
- Invoke the native method from Java

For example, our Java code could look like this:

```
package com.marakana.jniexamples;

public class Hello {
  public native void sayHi(String who, int times); //❶

  static { System.loadLibrary("HelloImpl"); } //❷

  public static void main (String[] args) {
    Hello hello = new Hello();
    hello.sayHi(args[0], Integer.parseInt(args[1])); //❸
  }
}
```

❶ ❸ The method `sayHi` will be implemented in C/C++ in separate file(s), which will be compiled into a library.

❷ The library filename will be called `libHelloImpl.so` (on Unix), `HelloImpl.dll` (on Windows) and `libHelloImpl.jnilib` (Mac OSX), but when loaded in Java, the library has to be loaded as `HelloImpl`.

16.4. JNI Development (C)

- We use the JDK `javah` utility to generate the header file `package_name_classname.h` with a function prototype for the `sayHi` method: `javac -d ./classes/ ./src/com/marakana/jniexamples/Hello.java` Then in the classes directory run: **javah -jni com.marakana.jniexamples.Hello** to generate the header file **com_marakana_jniexamples_Hello.h**

- We then create **com_marakana_jniexamples_Hello.c** to implement the **Java_com_marakana_jniexamples_Hello_sayHi** function

The file **com_marakana_jniexamples_Hello.h** looks like:

```
...
#include <jni.h>
...
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi
  (JNIEnv *, jobject, jstring, jint);
...
```

The file Hello.c looks like:

```
#include <stdio.h>
#include "com_marakana_jniexamples_Hello.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_Hello_sayHi(JNIEnv
*env, jobject obj, jstring who, jint times) {
  jint i;
  jboolean iscopy;
  const char *name;
  name = (*env)->GetStringUTFChars(env, who, &iscopy);
  for (i = 0; i < times; i++) {
    printf("Hello %s\n", name);
  }
}
```

16.5. JNI Development (Compile)

- We are now ready to compile our program and run it
    - The compilation is system-dependent

- This will create **libHelloImpl.so**, **HelloImpl.dll**, **libHelloImpl.jnilib** (depending on the O/S)

- Set **LD_LIBRARY_PATH** to point to the directory where the compiled library is stored

- Run your Java application

For example, to compile **com_marakana_jniexamples_Hello.c** in the "classes" directory (if your **.h** file and **.c** file are there) on Linux do:

```
gcc -o libHelloImpl.so -lc -shared \
    -I/usr/local/jdk1.6.0_03/include \
    -I/usr/local/jdk1.6.0_03/include/linux com_marakana_jniexamples_Hel
lo.c
```

On Mac OSX :

```
gcc -o libHelloImpl.jnilib -lc -shared \
    -I/System/Library/Frameworks/JavaVM.framework/Headers com_marakana_
jniexamples_Hello.c
```

Then set the `LD_LIBRARY_PATH` to the current working directory:

```
export LD_LIBRARY_PATH=.
```

Finally, run your application in the directory where your compiled classes are stored ("classes" for example):

```
java com.marakana.jniexamples.Hello Student 5
Hello Student
Hello Student
Hello Student
Hello Student
Hello Student
```

Note

Common mistakes resulting in `java.lang.UnsatisfiedLinkError` usually come from incorrect naming of the shared library (O/S-dependent), the library not being in the search path, or wrong library being loaded by Java code.

16.6. Type Conversion

- In many cases, programmers need to pass arguments to native methods and they do also want to receive results from native method calls
- Two kind of types in Java:
  - Primitive types such as `int`, `float`, `char`, etc
  - Reference types such as classes, instances, arrays and strings (instances of `java.lang.String` class)
- However, primitive and reference types are treated differently in JNI
  - Mapping for primitive types in JNI is simple

Table 3. JNI data type mapping in variables:

Java Language Type	Native Type	Description
boolean	jboolean	8 bits, unsigned
byte	jbyte	8 bits, signed

char	jchar	16 bits, unsigned
double	jdouble	64 bits
float	jfloat	32 bits
int	jint	32 bits, signed
long	jlong	64 bits, signed
short	jshort	16 bits, signed
void	void	N/A

- Mapping for objects is more complex. Here we will focus only on strings and arrays but before we dig into that let us talk about the native methods arguments

- JNI passes objects to native methods as opaque references
- Opaque references are C pointer types that refer to internal data structures in the JVM
- Let us consider the following Java class:

```
package com.marakana.jniexamples;

public class HelloName {
  public static native void sayHelloName(String name);

  static { System.loadLibrary("helloname"); }

  public static void main (String[] args) {
    HelloName hello = new HelloName();
    String name = "John";
    hello.sayHelloName(name);
  }
}
```

- The `.h` file would look like this:

```
...
#include <jni.h>
...
```

```
JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayH
elloName
  (JNIEnv *, jclass, jstring);
...
```

- A `.c` file like this one would not produce the expected result:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayH
elloName(JNIEnv *env, jclass class, jstring name){
    printf("Hello %s", name);
}
```

16.7. Native Method Arguments

- All native method implementation accepts two standard parameters:
  - `JNIEnv *env`: Is a pointer that points to another pointer pointing to a function table (array of pointer). Each entry in this function table points to a JNI function. These are the functions we are going to use for type conversion
  - The second argument is different depending on whether the native method is a static method or an instance method
    - Instance method: It will be a `jobject` argument which is a reference to the object on which the method is invoked
    - Static method: It will be a `jclass` argument which is a reference to the class in which the method is define

16.8. String Conversion

- We just talked about the `JNIEnv *env` that will be the argument to use where we will find the type conversion methods
- There are a lot of methods related to strings:
  - Some are to convert `java.lang.String` to `C` string: `GetStringChars` (Unicode format), `GetStringUTFChars` (UTF-8 format)
  - Some are to convert `java.lang.String` to `C` string: `NewString` (Unicode format), `NewStringUTF` (UTF-8 format)
  - Some are to release memory on C string: `ReleaseStringChars`, `ReleaseStringUTFChars`

Note

Details about these methods can be found at http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/functions.html

- If you remember the previous example, we had a native method where we wanted to display "Hello *name*":

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayH
elloName(JNIEnv *env, jclass class, jstring name) {
    printf("Hello %s", name); //❶
}
```

❶ This example would not work since the `jstring` type represents strings in the Java virtual machine. This is different from the C string type (`char *`)

- Here is what you would do, using UTF-8 string for instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_HelloName.h"

JNIEXPORT void JNICALL Java_com_marakana_jniexamples_HelloName_sayH
elloName(JNIEnv *env, jclass class, jstring name){
    const jbyte *str;
    str = (*env)->GetStringUTFChars(env, name, NULL); //❶
    printf("Hello %s\n", str);
    (*env)->ReleaseStringUTFChars(env, name, str); //❷
}
```

❶ This returns a pointer to an array of bytes representing the string in UTF-8 encoding (without making a copy)

❷ When we are not making a copy of the string, calling `ReleaseStringUTFChars` prevents the memory area used by the string to stay "pinned". If the data was copied, we need to call `ReleaseStringUTFChars` to free the memory which is not used anymore

- Here is another example where we would construct and return a `java.lang.String` instance:

```
#include <stdio.h>
#include "com_marakana_jniexamples_GetName.h"

JNIEXPORT jstring JNICALL Java_com_marakana_jniexamples_ReturnName_
GetName(JNIEnv *env, jclass class) {
    char buffer[20];
    scanf("%s", buffer);
    return (*env)->NewStringUTF(env, buffer);
}
```

16.9. Array Conversion

- Here we are going to focus on primitive arrays only since they are different from objects arrays in JNI
- Arrays are represented in JNI by the `jarray` reference type and its "subtypes" such as `jintArray` ⇒ A `jarray` is not a C array!
- Again we will use the `JNIEnv *env` parameter to access the type conversion methods
    - `Get<Type>ArrayRegion`: Copies the contents of primitive arrays to a preallocated C buffer. Good to use when the size of the array is known
    - `Get<Type>ArrayElements`: Gets a pointer to the content of the primitive array
    - `New<Type>Array`: To create an array specifying a length
- We are going to see an example of how to read a Java primitive array in the native world
- First, this would be your Java program:

```
package com.marakana.jniexamples;

public class ArrayReader {
    private static native int sumArray(int[] arr); //❶
    public static void main(String[] args) {
        //Array declaration
        int arr[] = new int[10];
        //Fill the array
        for (int i = 0; i < 10; i++) {
            arr[i] = i;
        }
        ArrayReader reader = new ArrayReader();
        //Call native method
        int result = reader.sumArray(arr); //❷
        System.out.println("The sum of every element in the array i
s " + Integer.toString(result));
    }
    static {
        System.loadLibrary("arrayreader");
    }
}
```

❶ This method will return the sum of each element in the array
❷

- After running `javah`, create your `.c` file that would look like this:

```
#include <stdio.h>
#include "com_marakana_jniexamples_ArrayReader.h"

JNIEXPORT jint JNICALL Java_com_marakana_jniexamples_ArrayReader_su
mArray(JNIEnv *env, jclass class, jintArray array) {
    jint *native_array;
    jint i, result = 0;
    native_array = (*env)->GetIntArrayElements(env, array, NULL); /
* ❶ */
    if (native_array == NULL) {
        return 0;
    }
    for (i=0; i<10; i++) {
        result += native_array[i];
    }
    (*env)->ReleaseIntArrayElements(env, array, native_array, 0);
    return result;
}
```

❶ We could also have used `GetIntArrayRegion` since we exactly know the size of the array

16.10. Throwing Exceptions In The Native World

- We are about to see how to throw an exception from the native world
- Throwing an exception from the native world involves the following steps:
    - Find the exception class that you want to throw
    - Throw the exception
    - Delete the local reference to the exception class
- We could imagine a utility function like this one:

```
void ThrowExceptionByClassName(JNIEnv *env, const char *name, const
char *message) {
  jclass class = (*env)->FindClass(env, name); //❶
  if (class != NULL) {
      (*env)->ThrowNew(env, class, message); //❷
  }
  (*env)->DeleteLocalRef(env, class); //❸
}
```

❶ Find exception class by its name

❷ Throw the exception using the class reference we got before and the message for the exception

❸ Delete local reference to the exception class

- Here would be how to use this utility method:

```
ThrowExceptionByClassName(env,"java/lang/IllegalArgumentExceptio
n","This exception is thrown from C code");
```

16.11. Access Properties And Methods From Native Code

- You might want to modify some properties or call methods of the instance calling the native code

- It always starts with this operation: Getting a reference to the object class by calling the `GetObjectClass` method
- We are then going to get instance field id or an instance method id from the class reference using `GetFieldID` or `GetMethodID` methods
- For the rest, it differs depending on whether we are accessing a field or a method

- From this Java class, we will see how to call its methods or access its properties in the native code:

```
package com.marakana.jniexamples;

public class InstanceAccess {
    public String name; //❶

    public void setName(String name) { //❷
        this.name = name;
    }

    //Native method
    public native void propertyAccess(); //❸
    public native void methodAccess(); //❹

    public static void main(String args[]) {
        InstanceAccess instanceAccessor = new InstanceAccess();
        //Set the initial value of the name property
        instanceAccessor.setName("Jack");
        System.out.println("Java: value of name = \""+ instanceAcce
ssor.name +"\"");
        //Call the propetyAccess() method
        System.out.println("Java: calling propertyAccess() metho
d...");
        instanceAccessor.propertyAccess(); //❺
        //Value of name after calling the propertyAccess() method
        System.out.println("Java: value of name after calling prope
rtyAccess() = \""+ instanceAccessor.name +"\"");
        //Call the methodAccess() method
        System.out.println("Java: calling methodAccess() metho
d...");
        instanceAccessor.methodAccess(); //❻
        System.out.println("Java: value of name after calling metho
dAccess() = \""+ instanceAccessor.name +"\"");
    }

    //Load library
    static {
        System.loadLibrary("instanceaccess");
    }
}
```

❶ Name property that we are going to modify along this code execution

❷ This method will be called by the native code to modify the name property

❸ ❺ This native method modifies the name property by directly accessing the property

❹
❻ This native method modifies the name property by calling the Java method `setName()`

- This would be our C code for native execution:

```
#include <stdio.h>
#include "com_marakana_jniexamples_InstanceAccess.h"


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess
_propertyAccess(JNIEnv *env, jobject object){
    jfieldID fieldId;
    jstring jstr;
    const char *cString;

    /* Getting a reference to object class */
    jclass class = (*env)->GetObjectClass(env, object); /* ❶ */

    /* Getting the field id in the class */
    fieldId = (*env)->GetFieldID(env, class, "name", "Ljava/lang/St
ring;"); /* ❷ */
    if (fieldId == NULL) {
        return; /* Error while getting field id */
    }

    /* Getting a jstring */
    jstr = (*env)->GetObjectField(env, object, fieldId); /* ❸ */

    /* From that jstring we are getting a C string: char* */
    cString = (*env)->GetStringUTFChars(env, jstr, NULL); /* ❹ */
    if (cString == NULL) {
        return; /* Out of memory */
    }
    printf("C: value of name before property modification = \"%s
\"\n", cString);
    (*env)->ReleaseStringUTFChars(env, jstr, cString);

    /* Creating a new string containing the new name */
    jstr = (*env)->NewStringUTF(env, "Brian"); /* ❺ */
    if (jstr == NULL) {
        return; /* Out of memory */
    }
    /* Overwrite the value of the name property */
    (*env)->SetObjectField(env, object, fieldId, jstr); /* ❻ */
}


JNIEXPORT void JNICALL Java_com_marakana_jniexamples_InstanceAccess
_methodAccess(JNIEnv *env, jobject object){
    jclass class = (*env)->GetObjectClass(env, object); /* ❼ */
    jmethodID methodId = (*env)->GetMethodID(env, class, "setName",
"(Ljava/lang/String;)V"); /* ❽ */
    jstring jstr;
    if (methodId == NULL) {
        return; /* method not found */
    }
    /* Creating a new string containing the new name */
    jstr = (*env)->NewStringUTF(env, "Nick"); /* ❾ */
    (*env)->CallVoidMethod(env, object, methodId, jstr); /* ❿ */
}
```

This is getting a reference to the object class

Gets a field Id from the object class, specifying the property to get and the internal type. you can find information on the jni type there: http://download.oracle.com/javase/6/docs/technotes/guides/jni/spec/types.html
This will return the value of the property in the native type: here a jstring

We need to convert the jstring to a C string

This creates a new java.lang.String that is going be use to change the value of the property
This sets the property to its new value

Gets a method id from the object class previously obtained, specifying the name of the method along with its signature. There is a very useful java tool that you can use to get the signature of a method: `javap -s -p ClassName` for instance `javap -s -p InstanceAccess`
This creates a new java.lang.String that we are going to use as an argument when calling the java method from native code
Calling `CallVoidMethod` since the Java method return type is `void` and we are passing the previously created `jstring` as a parameter

Prev Next
Home | ToC

Developer Training

Free Resources

About

BACK TO TOP