Java Native Interface in OS/2

Jarda Kačer
jkacer@kiv.zcu.cz

Czech Warpstock 2004,
Pec pod Sněžkou, CZ
2004-06-27

Presentation Outline

- Reasons to Use JNI
- Technology Overview
- “Hello Warpstock” Tutorial
- OS/2 JNI Issues and Solutions
- Java Types vs. C Types
- Calling Java Code Back from C Code
- Accessing Fields
- Results
- Links and Resources

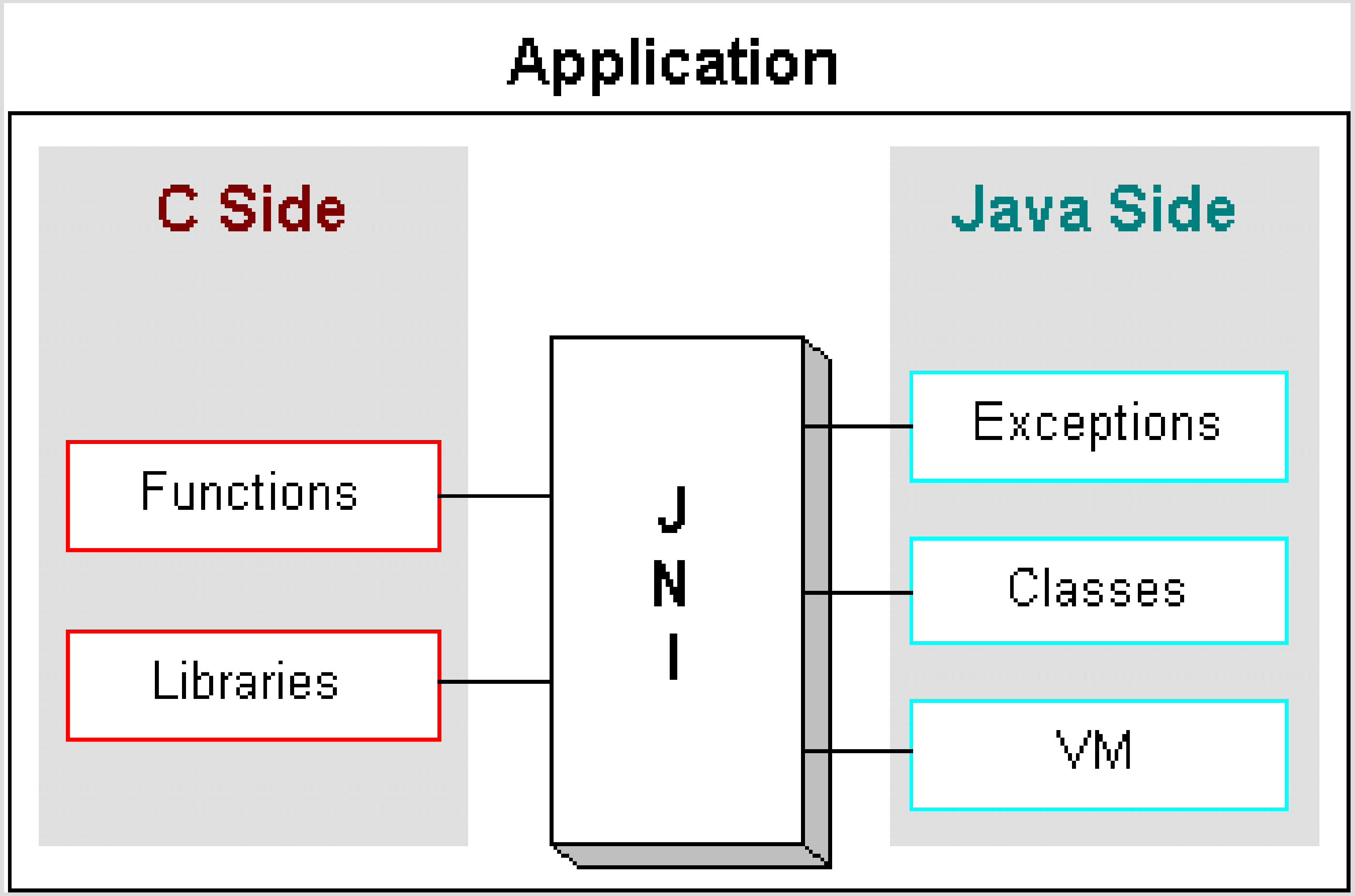
Reasons to Use JNI

- A feature you need is not available in Java
- You already have your application written in another language and need to call it from Java
- A piece of time-critical code with properties that Java is not able to guarantee

Technology Overview (1)

- JNI = A “glue” between Java code and external libraries (written mainly in C or C++)
- Two types of calls:
  - From Java to a library
  - From a library back to Java
- The library format is platform-dependent
  - DLL on OS/2 and Win, so on Linux and Solaris
- On the library side, you can work with:
  - Objects
  - Classes
  - Exceptions
  - Threads

Technology Overview (2)



Hello Warpstock Tutorial (1)

- A Hello-World-like Java program that uses JNI to call a function in OS/2 DLL
- The function prints out “Hello Czech Warpstock 2004!”
- Developed with Golden Code Java 1.4.1 and Open Watcom 1.2
- Based on Java Tutorial, JNI trail

Hello Warpstock Tutorial (2)

- Steps:
  1. Write MyClass.java
  2. Compile MyClass.java to MyClass.class
  3. Generate header file MyClass.h
  4. Write implementation of native functions to MyClassImpl.c
  5. Compile MyClassImpl.c to HelloLib.dll
  6. Run MyClass.class

Hello Warpstock Tutorial (3)

Step 1: Write MyClass.java

```
class HelloWarpstock
{
    public native void sayHello();

    static
    {
        // Load HelloLib.DLL
        // Max 8 characters !!!
        System.loadLibrary("HelloLib");
    } // static

    public static void main(String[] args)
    {
        HelloWarpstock hw;

        hw = new HelloWarpstock();
        hw.sayHello();
    } // main
} // class HelloWarpstock
```

Hello Warpstock Tutorial (4)

Step 2: Compile MyClass.java into MyClass.class

```
javac MyClass.java
```

Hello Warpstock Tutorial (5)

Step 3: Generate header file MyClass.h

```
javah -jni MyClass.java
```

```
#include <jni.h>
#ifndef _Included_HelloWarpstock
#define _Included_HelloWarpstock

#ifdef __cplusplus
extern "C" {
#endif
JNIEXPORT void __export JNICALL Java_HelloWarpstock_sayHello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

Hello Warpstock Tutorial (6)

Step 4: Write implementation of native functions into MyClassImpl.c

```
#include <jni.h>
#include "HelloWarpstock.h"
#include <stdio.h>

JNIEXPORT void __export JNICALL
Java_HelloWarpstock_sayHello(JNIEnv *env, jobject obj)
{
    printf("Hello Czech Warpstock 2004!\n");
    return;
}
```

Hello Warpstock Tutorial (7)

Step 5: Compile MyClassImpl.c into HelloLib.dll

A complicated task, see the OS/2 issues later.

```
wmake
```

It actually does both compilation into MyClassImpl.c and linking into HelloLib.dll
See makefile and LinkOptions.lnk

Hello Warpstock Tutorial (8)

Step 6: Run MyClass.class

```
java MyClass
```

OS/2 JNI Issues and Solutions

- The DLL's name must fit into 8+3 letters!
  - You will get UnsatisfiedLinkerError + Error 123 in GC Java if it does not, similarly in InnoTek Java
  - OS/2 limitation
- Functions must be exported from the DLL!
  - JNIEXPORT should do it, but doesn't
  - Always check jni_md.h in Java\include\os2
  - Solution for OpenWatcom: Add __export by hand after return type (2 underscores)
- Functions will be called by the system
  - _System is OK for OW (expanded from JNICALL)

Java Types vs. C Types

- Java types cannot be used directly in C code
- Mapping stored in jni.h in your Java distribution
- Different types for C and C++
- Two kinds of type:
  - Primitive types
  - Objects

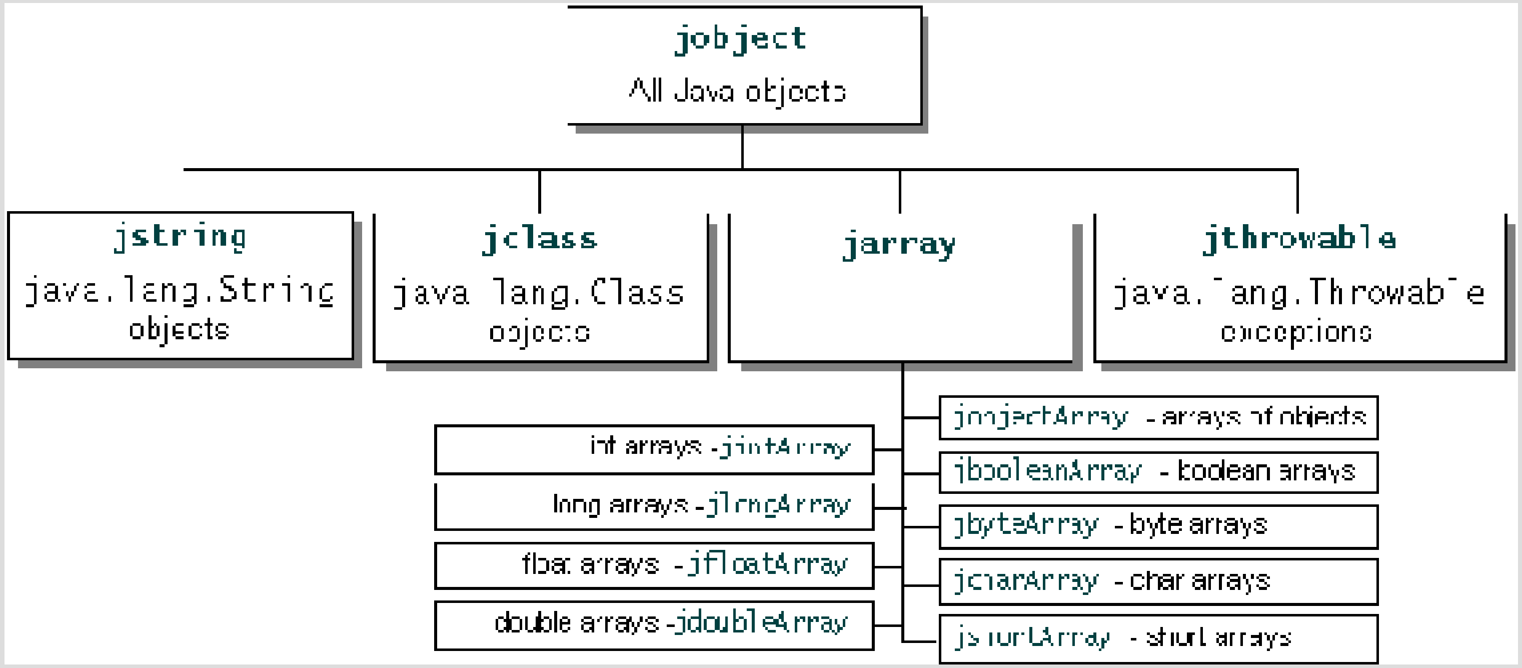
Primitive Types

- boolean – jboolean
- byte – jbyte
- char – jchar
- short – jshort
- int – jint
- long – jlong
- float – jfloat
- double – jdouble
- void – void

Objects (1)

- Object – jobject, root of everything
- String – jstring
- Class – jclass
- Trowable – jtrowable
- [] <type> – j<type>Array
- Every function get JNIEnv* and jobject:
  - JNIEnv* env: env is a pointer to the Java environment, cannot be shared among different threads
  - jobject this: this is a pointer to the instance that invoked the method

Objects (2)



jobject
All Java objects
jstring
java.lang.String objects
jclass
java.lang.Class objects
jarray
jthrowable
java.lang.Throwable exceptions
int arrays - jintArray
long arrays - jlongArray
float arrays - jfloatArray
double arrays - jdoubleArray
jobjectArray - arrays of objects
jbooleanArray - boolean arrays
jbyteArray - byte arrays
jcharArray - char arrays
jshortArray - short arrays

Working with Strings

- GetStringChars takes the Java string and returns a pointer to an array of Unicode characters
- ReleaseStringChars releases the pointer to the array of Unicode characters
- NewString constructs a new java.lang.String from an array of Unicode characters
- GetStringLength returns the length of a string that is comprised of an array of Unicode characters
- GetStringUTFLength returns the length of a string if it is represented in the UTF-8 format

Working with Arrays

- Get<type>ArrayElements returns the elements and pins down the array
- Release<type>ArrayElements unpins the memory
- Get/Set<type>ArrayRegion
- GetObjectArrayElement
- SetObjectArrayElement

Calling Java Code

- jclass GetObjectClass(env, obj);
- jmethodID GetMethodID(env, cls, "name", "signature");
  - Signature is important, methods can be overloaded
- CallVoidMethod(env, obj, mid, params);
- Call<type>Method(env, obj, mid, params);
- Similarly for static methods
- Example later

Method Signatures

- (argument-types)return-type
- Z – boolean
- B – byte
- C – char
- S – short
- I – int
- J – long
- F – float
- D – double
- Lfully-qualified-class – fully-qualified-class
- [type – type[]

Calling Java Code Example

```
JNIEXPORT void JNICALL
Java_Callbacks_nativeMethod(JNIEnv *env, jobject obj, jint depth)
{
 jclass cls = (*env)->GetObjectClass(env, obj);
 jmethodID mid = (*env)->GetMethodID(env, cls, "callback", "(I)V");
 if (mid == 0)
 {
  return;
 }
 printf("In C, depth = %d, about to enter Java\n", depth);
 (*env)->CallVoidMethod(env, obj, mid, depth);
 printf("In C, depth = %d, back from Java\n", depth);
}
```

Accessing Fields

- Two steps: First get its ID, then its value
- Get ID:
  - GetStaticFieldID(env, cls, "name", "signature");
  - GetFieldID(env, cls, "name", "signature");
- Get value:
  - GetStatic<type>Field(env, cls, fid);
  - Get<type>Field(env, obj, fid);
- Signatures are the same as when calling methods
- If unsure, run javap -s -p MyClass

JNI and Multithreading

- Synchronization must be supported in JNI
- MonitorEnter(env, obj);
- MonitorExit(env, obj);
- wait(), notify(), notifyAll(): Not directly supported, can be performed via method calls, as any other method

What We Did Not Talk About

- Exception throwing, catching, handling
- The problem of local and global references, their scope of validity
- JNI and C++
- Invoking the JVM, attaching native threads

Results

- You will not probably need to use JNI for ordinary applications
- JNI may come in handy in special cases
- It's good to know the tricks on OS/2
- No fear, it's just a little bit more difficult than ordinary C programming :-)

Links and Resources

- http://java.sun.com/docs/books/tutorial/native1.1/index.html
- http://java.sun.com/j2se/1.4.2/docs/guide/jni/
- http://java.sun.com/developer/codesamples/jni.html
- http://java.sun.com/docs/books/jni/
- http://home.t-online.de/home/howlingmad/watcom_tip_en.html
- See the screenshots