The JAVA NATIVE INTERFACE

an introduction

Alison Lloyd

JNI isn't difficult to use…

…and it doesn't have to be complicated!

But it can be a very powerful and useful tool…

…when used correctly

Contents

- What is it and what can it do?
- When should or shouldn't you use it?
- Basic JNI application overview
- The 4-steps to JNI application happiness
- Some JNI specifics
  - Naming and typing
  - Some helper functionality
- Calling Java code from native code

But wait, there's more!

GOTCHAS

- Demos
- Further reading

What is JNI?

- Standard part of Java
- Allows you to integrate native code (written in, say, C) into Java applications
- Can also be used to embed a JVM into a native application
- Provides a standard means of interaction between Java code and native code

5

So what can JNI do?

- Execute native code from within Java
  - Call native methods
- Execute Java code from within native code
  - Catch & throw exceptions
  - Call methods
  - Use objects
- Embed the JVM in native application (via the Invocation API)

6

Why use JNI?

- Direct hardware access / support
- Reuse existing libraries
- Time-critical code / operations
- Better support for something in another language…

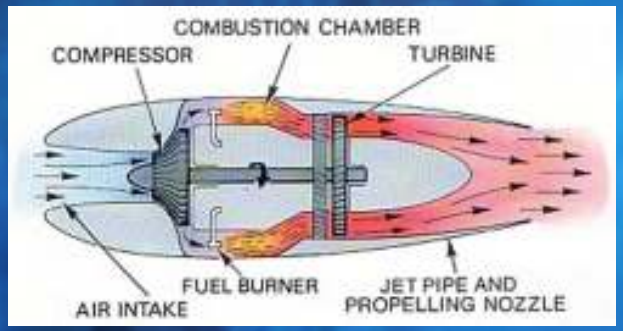
7

Why not use JNI?

- “The system absolutely has to be written in Java!”
- Using a platform-dependant library makes your application (more) platform-dependant
- Codebase becomes more complicated
  - More opportunity for memory leaks, etc.

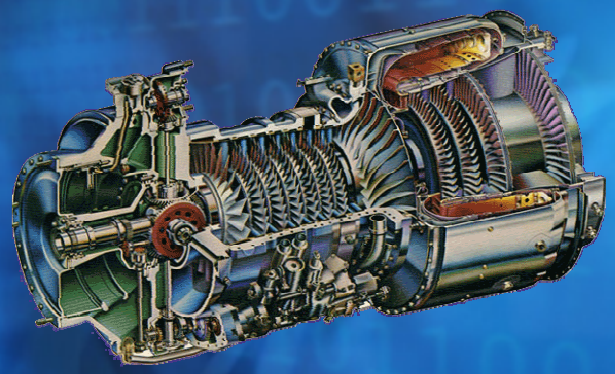
8

Theoretical complexity…



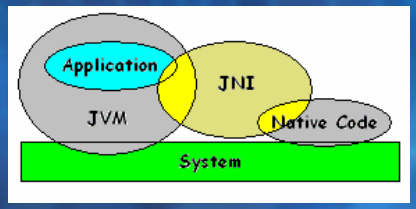
9

Real-world Complexity…



10

A basic JNI application



- JNI provides the link between the application and some native code
- Also allows the native code to access the JVM (and thus the application)

11

Creating a JNI application

1. Decide what functionality needs to be in native code
2. Write & compile Java wrapper class
3. Create native (C / C++) header using *javah*
4. Import header into new DLL and populate functions

12

1. Design

- Try to keep native code to a minimum
- Avoid passing platform-dependant stuff (if possible)
- Clearly divided functionality (one function per action)
- Might need several native libraries

13

2. Create Java wrapper class

- Normal Java class, plus:
  - `native` methods
  - `static` block that loads the native library via a call to `System.loadLibrary`
- Native methods have no implementation (like abstract methods)
- Compile class as normal

14

3. Generate native header

- Use *javah*, which comes with the JDK:
  javah –jni <wrapper class name>
  for example,
  javah –jni MyPkg.MyWrapper
- Note package name (if used) must be specified!
- This produces <wrapper class name>.h

15

4. Write native code

- Import header
- Each `native` method in the wrapper class should have a munged equivalent
- Implement the functions
  - The `JNIEnv` object can be used to access various helpful bits of the JNI
  - Other parameters are converted to “C” types

16

4a. Native compiler setup

- Need to add JNI headers to include path:
  - `<java install>\include` AND
  - `<java install>\include\<platform>`
- For example:
  - `C:\jdk1.5.0_15\include`
  - `C:\jdk1.5.0_15\include\win32`

17

Demo

18

JNI name munging

- Native function name is created from:
  - The prefix *Java_*
  - Mangled fully-qualified class name
  - A separator ("_")
  - Mangled method name
  - For overloaded native methods, two underscores followed by the mangled argument signature
- May overload non-native methods

19

The JNI Environment object

- `JNIEnv` object used to access JNI functionality
- Passed as first two parameters to all native functions
- Examples of use:
  - Retrieving array elements
  - Getting strings
  - Accessing Invocation API
- See chapters 4 and 5 of spec

20

Playing ‘hunt the library’

- The argument passed to `System.loadLibrary` is converted to platform naming convention, for example:
  - MyLib.dll for Win32
  - libMyLib.so for Solaris
- The library must be somewhere the JVM can find it
  - System search path
  - Usually in the same directory as the application

GOTCHA

21

String operations

- Java uses UTF – need to convert strings
  - `GetStringUTFChars`
  - MUST ReleaseStringUTFChars
- Can also create new `java.lang.Strings`, get region encoding, etc.
- Any created object (or other allocated memory) must be freed when you’re done with it!

GOTCHA

22

String helper functions

23

Array operations

- Primitive arrays vs. Object arrays
- Primitives:
  - `GetXXXArrayRegion`
  - `GetXXXArrayElements`, `ReleaseXXXArrayElements`
- Objects:
  - `NewObjectArray`
  - `Get / SetObjectArrayElement`
  - `FindClass`

24

Type signatures

- Uses JVM type signature representation
- Single letters for primitive types, or fully qualified class names
- `( arg-types ) ret-type` for a method

  e.g. `long foo (int n, String s)`

  gives `(ILjava/lang/String;)J`

25

Calling Java code from JNI

- Create Java wrapper class and native header as above
- Native code needs to know class, method name, and method signature of Java code it wants to call
- `JNIEnv->GetMethodID` and `JNIEnv->CallXXXMethod`
- `JNIEnv->GetStaticMethodID` and `JNIEnv->CallStaticXXXMethod`

26

Accessing fields of objects

- Basically the same as calling Java methods:

  `GetFieldID()`

  `GetXXXField()` and `SetXXXField()`
- Need an instantiated class, field name and signature
- Can't be used to get length of array – use `GetArrayLength()`

27

Demo

28

Gotchas

- Package name when creating native headers
- Memory leaks when working with strings
- Multiple instances of the same library
- `System.load` vs `System.loadLibrary`

29

Recommended reading

- JNI Specification (Java 5):
  http://java.sun.com/j2se/1.5.0/docs/guide/jni/spec/jniTOC.html
- Sun JNI tutorial:
  http://java.sun.com/docs/books/tutorial/information/download.html
- *Java Native Interface: Programmer's Guide and Specification*:
  http://java.sun.com/docs/books/jni/index.html

30

31