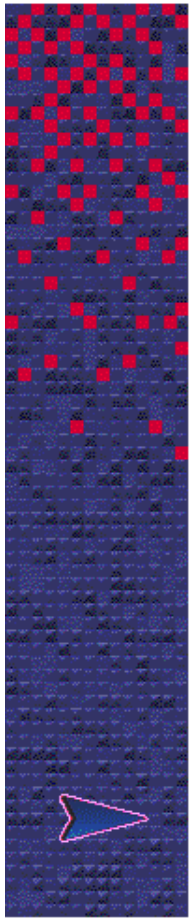


Java™ Native Interface Technology Programming

Sheng Liang, Staff Engineer
Anand Palaniswamy, Member of Tech Staff
Sun Microsystems, Inc.

Other Available Formats | JavaOne Home Page

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Why?

Leverage legacy code

- Graphics engines, database backends

Embed Java™ VM in native apps

- Web browsers and servers

Java™ Native Interface (JNI) Technology

A two-way glue

Also...

JNI is:

- **The standard**
- **VM independent**
- **Efficient**

Using JNI means:

- **No longer 100% Pure Java™ Certified!**
- **No type safety guarantees!**

Status

Introduced in JDK™ 1.1 Software

Minor enhancements in JDK 1.2

Used in:

- **All JDK 1.2 native methods**
- **Project Activator**
- **Java Media Framework**

Move to JNI!

Outline

- Introduction
  - Native Methods
  - Invocation Interface
- Programming Techniques
  - Patterns
  - Caching IDs
- Common Problems
  - Green Threads
  - Exceptions
  - Refs

Native Method Example

```
class UNIX {
  native static int chmod(String path, int mode);
  ...
}
```

Usage:

```
UNIX.chmod("/home/sl/mbox", 0600);
```

Implementation

Native method:

```
class UNIX {
  native static int chmod(String path, int mode);
```

C++ code:

```
extern "C" jint Java_UNIX_chmod (
  JNIEnv *env, jclass c, jstring path, jint mode)
{
  jint result;
  const *cpath = env->GetStringUTFChars(path);
  if (cpath == NULL) return 0; // out of memory
  result = chmod(cpath, mode);
  env->ReleaseStringUTFChars(path, cpath);
  return result;
}
```

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Invocation Interface

VM shipped as a shared library

Embeddable in native applications

Invocation Example

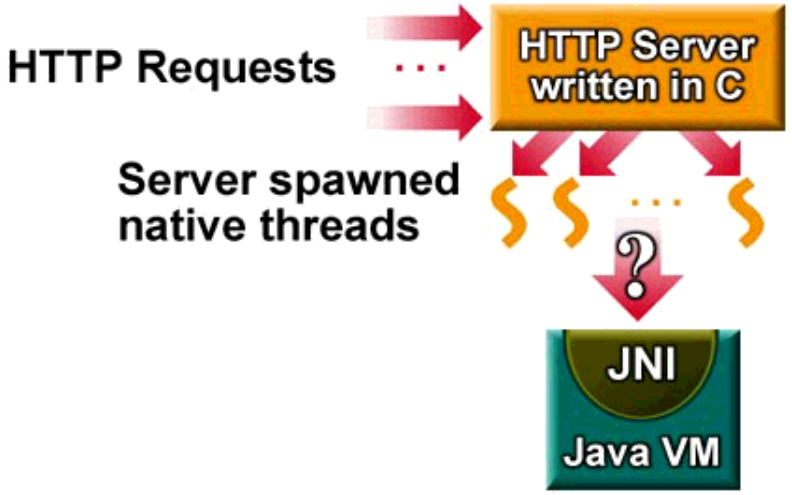
Invoke the VM and call `Main.run`:

```
JavaVM *jvm;
JNIEnv *env;
JNI_CreateJavaVM(&jvm, &env, &args);

jclass cls = env->FindClass("Main");
jmethodID mid =
    env->GetStaticMethodID(cls, "run", "()V");
env->CallStaticVoidMethod(cls, mid);

jvm->DestroyJavaVM();
```

Add more error checking in your code!

Multi-threading Support

Example: a multi-threaded HTTP server



Attach Native Threads

Attach to `jvm` and call `Service.run`:

```
JNIEnv *env;
jvm->AttachCurrentThread(&env, 0);

jclass cls = env->FindClass("Service");
jmethodID mid =
    env->GetStaticMethodID(cls, "run", "()V");
env->CallStaticVoidMethod(cls, mid);

jvm->DetachCurrentThread();
```

Again, add error checking in your code!

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Rules of Thumb

Keep interactions with native code simple

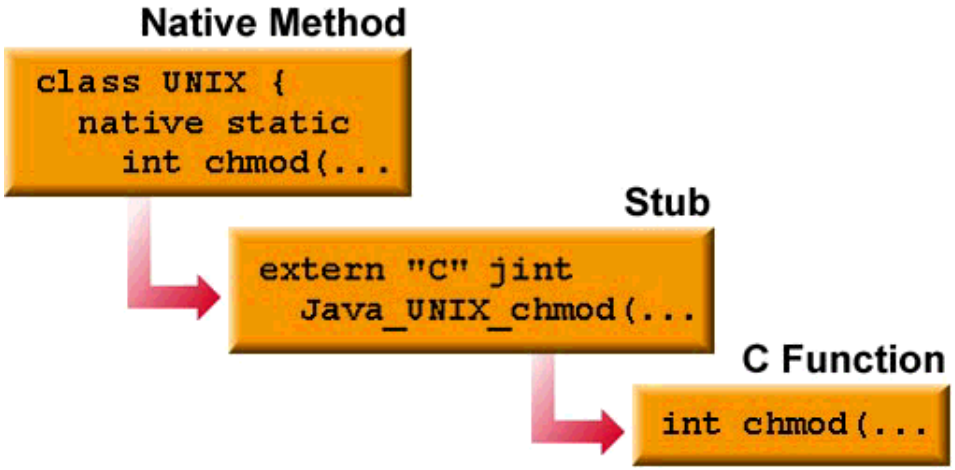
Do not use JNI to do Java™ programming in native code

Useful Patterns

1-1 Mapping

Shared Stubs

1-1 Mapping


Native Method
class UNIX {
native static
int chmod(...
Stub
extern "C" jint
Java_UNIX_chmod(...
C Function
int chmod(...

Shared Stubs

Shared stub `callInt` dispatches to many C functions

Given

```
class CFunc {
  static native CFunc find(String lib, String f);
  native int callInt(Object[] args);
}
```

we can implement `chmod` as:

```
class UNIX {
  static int chmod(String path, int mode) {
    CFunc cfunc = CFunc.find("libc.so", "chmod");
    Object[] args = {path, new Integer(mode)};
    return cfunc.callInt(args);
  }
}
```

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Name Lookup Is Slow

Native method:

```
class File {
  int fd;
  native byte readByte();
}
```

C++ stub:

```
jbyte Java_File_readByte(JNIEnv *env, jobject self)
{
  jbyte b;
  jclass cls = env->GetObjectClass(self);
  jfieldID id = env->GetFieldID(cls, "fd", "I");
  if (id == NULL) return 0;
  int fd = env->GetIntField(self, id);
  read(fd, &b, sizeof(jbyte));
  return b;
}
```

Cache Your IDs!

Native method:

```
class File {
  static native void initIDs();
  static { initIDs(); }
  int fd;
  native byte readByte();
}
```

C++ stub:

```
static jfieldID File_fd_ID;
jint Java_File_initIDs(JNIEnv *env, jclass cls) {
  File_fd_ID = env->GetFieldID(cls, "fd", "I");
}
jbyte Java_File_readByte(JNIEnv *env, jobject self) {
  jbyte b;
  int fd = env->GetIntField(self, File_fd_ID);
  read(fd, &b, sizeof(jbyte));
  return b;
}
```

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Green vs. Native Threads

Two threads packages on Solaris

- User level green threads (default)
- OS native threads

Green threads have limitations

- Do not work well with some native libraries
- Not embeddable

Outline

- Introduction
  - Native Methods
  - Invocation Interface
- Programming Techniques
  - Patterns
  - Caching IDs
- Common Problems
  - Green Threads
  - Exceptions
  - Refs

Check For Exceptions!

Recall checks in earlier examples

```
class UNIX {
  ...
  char *cpath = env->GetStringUTFChars(path);
  if (cpath == NULL) return 0;
  ...
class File {
  ...
  jfieldID id = env->GetFieldID(cls, "fd", "I");
  if (id == NULL) return 0;
  ...
```

Unchecked exceptions make your applications less robust

Outline

- **Introduction**
  - Native Methods
  - Invocation Interface
- **Programming Techniques**
  - Patterns
  - Caching IDs
- **Common Problems**
  - Green Threads
  - Exceptions
  - Refs

Local Refs

Freed automatically on native method return

Invalid across threads

This is wrong:

```
jint Java_some_Clazz_foo(JNIEnv *env, ...) {
  static jclass otherCls = NULL;
  if (otherCls == NULL) {
    otherCls = env->FindClass("other/Clazz");
    if (otherCls == NULL) return 0;
  }
  ... /* use otherCls */
}
```

Use Global Refs

This is correct:

```
jint Java_some_Clazz_foo(JNIEnv *env, ...) {
  static jclass otherCls = NULL;
  if (otherCls == NULL) {
    otherCls = env->FindClass("other/Clazz");
    if (otherCls == NULL) return 0;
    otherCls = env->NewGlobalRef(otherCls);
    if (otherCls == NULL) return 0;
  }
  ... /* use otherCls */
}
```

Don’t forget to delete them eventually!

Excessive Local Ref Creation

Delete if necessary

```
jint len = env->GetArrayLength(env, arr);
for (i = 0; i < len; i++) {
  jobject x = env->GetObjectArrayElement(arr, i);
  ... /* process x */
  env->DeleteLocalRef(x);
}
```

New in JDK™ 1.2 software

- 16 local refs guaranteed
- `EnsureLocalCapacity`
- `-verbose:jni` option

End Notes

Code for this talk

`http://java.sun.com/products/jdk/faq/jnifaq.html`

Comments and general feedback

`jni@java.sun.com`

Support through Java℠ Developer Connection℠

`http://java.sun.com/jdc`

Java™ Series book on JNI forthcoming