Java Native Interface

CS587x Lecture
Department of Computer Science
Iowa State University

Introduction

- What is native method and JNI
- Why we use JNI
- How to use JNI
  - Embedding C in Java
  - Using Java features from C
  - Embedding the VM

What is Native Method

- Functions written in a language other than Java
- They could be C, C++, or even assembly

What is JNI

- Java interface to non-Java code. It is Java's link to the "outside world"
  - Native methods are compiled into a dynamic link library (.dll, .so, etc.)
  - Java Virtual Machine loads and links the library into the process that calls the native methods
- Part of the Java Developer Kit(JDK), serves as a glue between java side and native side of an application
  - Allows Java code that runs inside a Java Virtual Machine (JVM) to interoperate with applications and libraries written in other programming languages, such as C, C++, and assembly

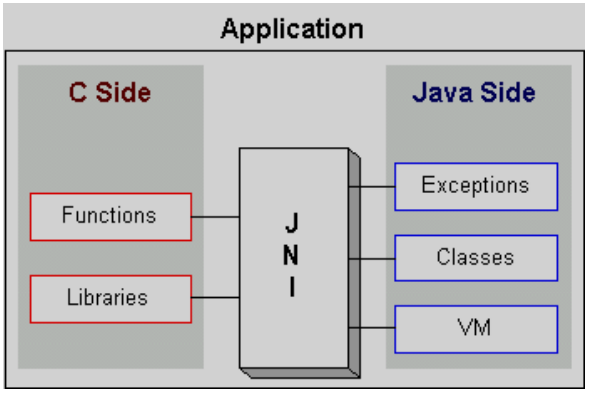
Why Use JNI

- Some functionality are not provided by java
  - Low-level drives/devices specific to OS
- Increase performance by implementing rigorous tasks in native language
  - Java is slow in general and may not be suitable for some functions (e.g., image compression and decompression)
- Try to use existing legacy library and could not afford to rewrite it in java
  - JNI can be used as a wrapper of these legacy codes
  - Can slowly migrate legacy code to a newer platform
- Improve efficiency of integration
  - TCP/IP sockets can be used, but there are overhead in data transmission

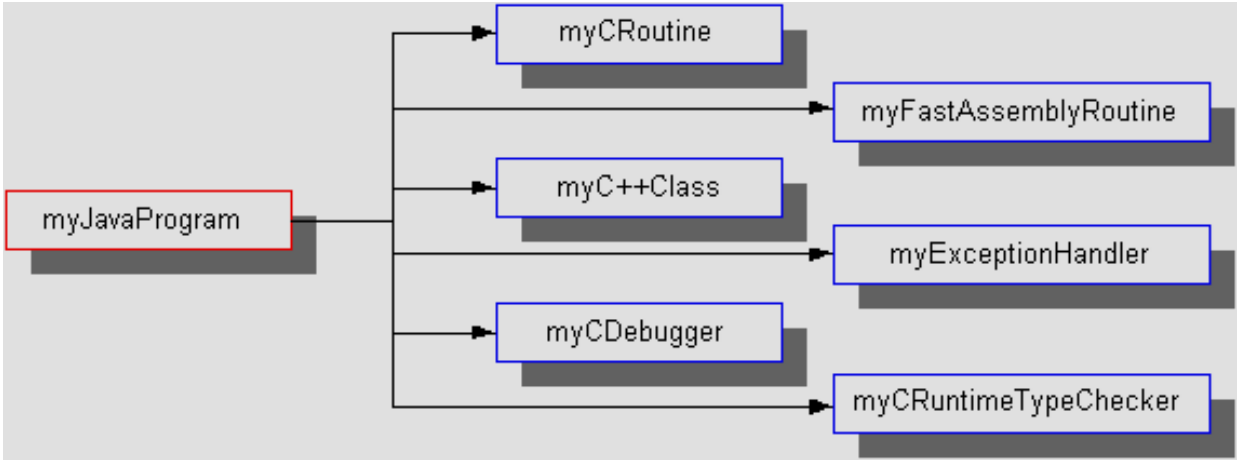
Justification

- Pros:
  - Reuse: allows access to useful native code
  - Efficiency: use best language for the task
- Cons:
  - Applets: doesn't work as they're mobile
  - Portability: native methods aren't portable
  - Extra work: **`javah`**, create shared native libs

JNI Overview



Access to Java world from native code



Access to native code from Java

Using The JNI

- Java calls C
  - Embedding C in Java
- C calls Java
  - Using Java features from C
  - Embedding the VM

Embedding C in Java

1. Declare the method using the keyword native, provide no implementation.
2. Make sure the Java loads the needed library
3. Run the javah utility to generate names/headers
4. Implement the method in C
5. Compile as a shared library

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

Generate JNI Header

- Compile HelloWorld.java
  - `javac HelloWorld.java`
- Generate HelloWorld.h
  - `javah HelloWorld`

HelloWorld.h

```
#include “jni.h”
/* Header for class HelloWorld */
#ifndef _Included_HelloWorld
#define _Included_HelloWorld
#ifdef __cplusplus
    extern “C” {
#endif

/*
 * Class: HelloWorld
 * Method: displayHelloWorld
 * Signature: ()V
 */
JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

HelloWorldImp.c

```
#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

```
class HelloWorld {
    . . .
        System.loadLibrary("hello");
    . . .
}
```

Compile the native code into a shared library:

- popeye (Linux)

```
cc -shared -I/usr/java/j2sdk1.4.1_04/include \
   -I/usr/java/j2sdk1.4.1_04/include/linux \
   HelloWorldImpl.c -o libhello.so
```

Run the Program

Command:

```
java HelloWorld
```

Result:

```
Hello World!
```

Possible exceptions:

```
java.lang.UnsatisfiedLinkError: no hello in shared
library path at
java.lang.Runtime.loadLibrary(Runtime.java) at
java.lang.System.loadLibrary(System.java) at
java.lang.Thread.init(Thread.java)
```

On popeye (Linux), do this:

```
LD_LIBRARY_PATH=./
export LD_LIBRARY_PATH
```

What a method can do?

```
class HelloWorld
{
    public native void displayHelloWorld(...);
     {
      1. Access a variable in an object
         • primitive: byte/short/int/long/char/boolean/double/float
         • complex: object, String, array of a primitive type or complex type
         • the variable can be static or non static
      2. Call a method in an object
         • static/non static
      3. ...
     }
}
```

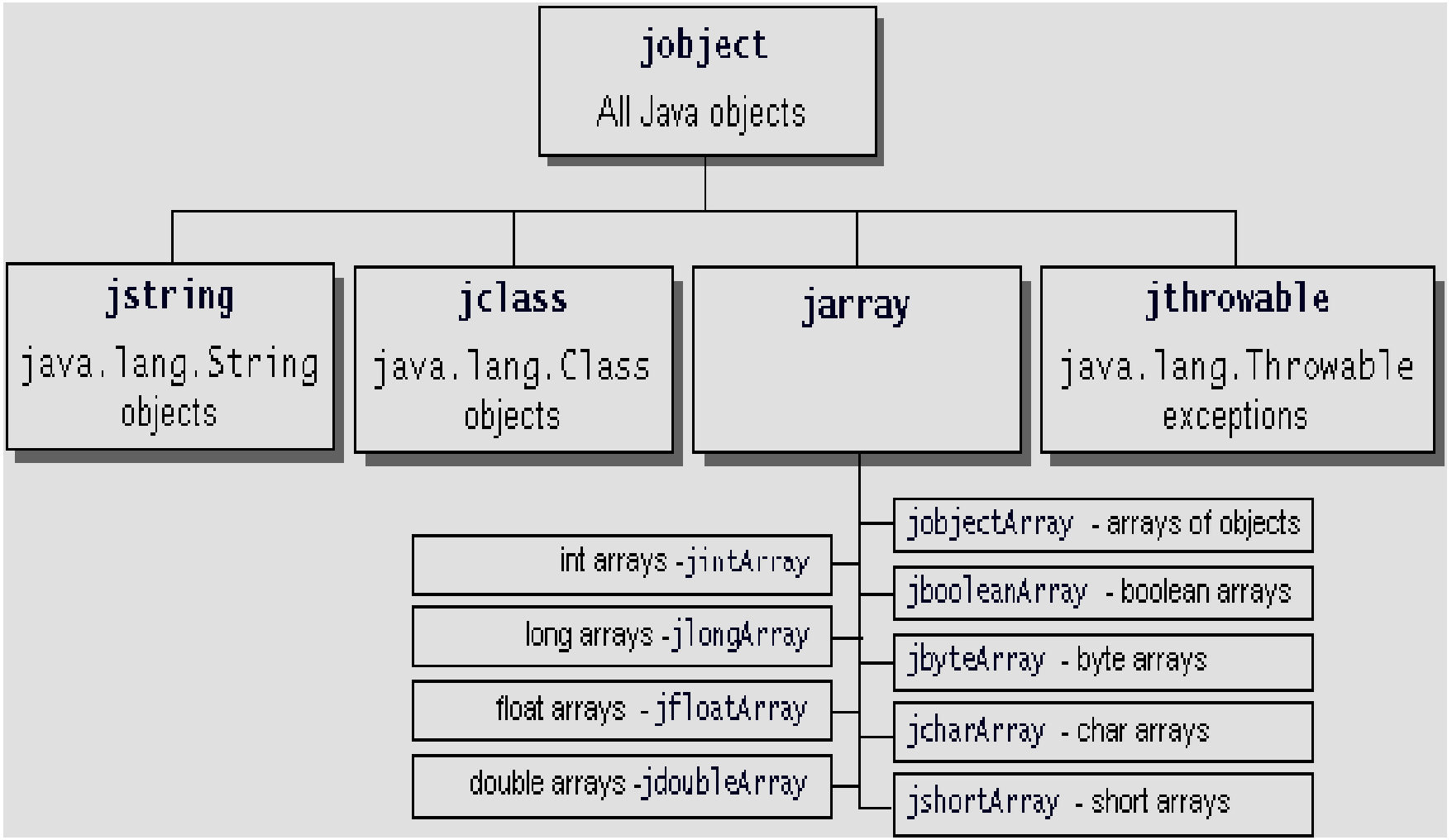
Primitive Types and Native Equivalents

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

Each element of Java language must have a corresponding native counterpart

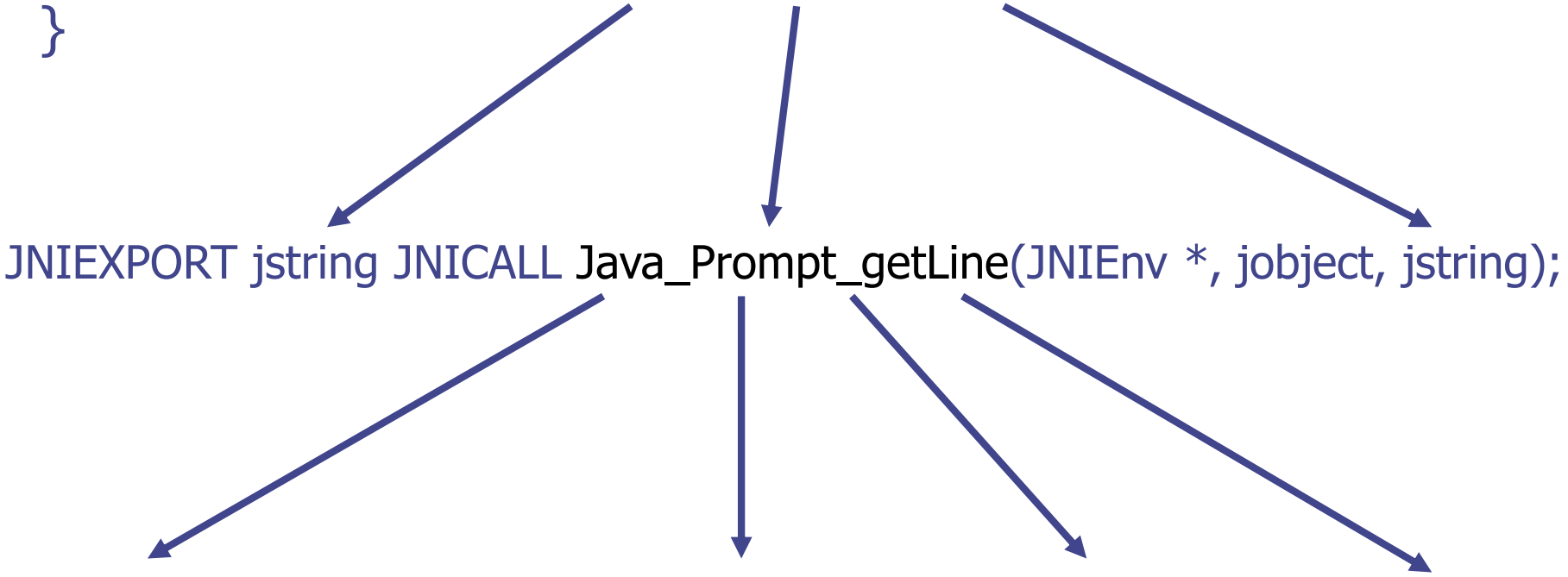
- Platform-specific implementation
- Generic interface to programmer

Object Types and Native Equivalents



Mapping Example

```
class Prompt
{
        private native String getLine(String prompt);
}
```



Accessing Java Strings

This jstring type is different from the regular C string type

```
/* Illegal */
JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
        printf("%s", prompt); ...
}
```

```
/* correct way */
JNIEXPORT jstring JNICALL Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
        const char *str = (*env)->GetStringUTFChars(env, prompt, 0);
        printf("%s", str);

        /* release the memory allocated for the string operation */
        (*env)->ReleaseStringUTFChars(env, prompt, str);

        char buf[128];

        scanf("%s", buf);
        return (*env)->NewStringUTF(env, buf); // return the string to caller
}
```

For the functions associated with JNI objects, go to web page:

http://docs.oracle.com/javase/7/docs/technotes/guides/jni/

Accessing Java Array

```
/* Illegal */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
      int i, sum = 0;
      for (i=0; i<10; i++)  {
            sum += arr[i];
      } ...
```

Accessing Java Array

```
/* Illegal */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
      int i, sum = 0;
      for (i=0; i<10; i++)  {
            sum += arr[i];
      } ...

/* correct way */
JNIEXPORT jint JNICALL Java_IntArray_sumArray(JNIEnv *env, jobject obj, jintArray arr)
{
      int i, sum = 0;

      /* 1. obtain the length of the array */
      jsize len = (*env)->GetArrayLength(env, arr);

      /* 2. obtain a pointer to the elements of the array */
      jint *body = (*env)->GetIntArrayElements(env, arr, 0);

      /* 3. operate on each individual primitive or jobjects */
      for (i=0; i<len; i++) {
            sum += body[i];
      }

      /* 4. release the memory allocated for array */
      (*env)->ReleaseIntArrayElements(env, arr, body, 0);
```

Accessing Java Member Variables

```
class FieldAccess
{
      static int si;  /* signature is “si” */
      String s;       /* signature is “Ljava/lang/String;";
}                     /* run javap -s -p FieldAccess to get the signature */
```

```
fid = (*env)->GetStaticFieldID(env, cls, "si", "I");  /* 1. get the field ID */
si = (*env)->GetStaticIntField(env, cls, fid);  /* 2. find the field variable */
(*env)->SetStaticIntField(env, cls, fid, 200);  /* 3. perform operation on the primitive*/

fid = (*env)->GetFieldID(env, cls, "s", "Ljava/lang/String;"); /* 1. get the field ID */
jstr = (*env)->GetObjectField(env, obj, fid); /* 2. find the field variable */
jstr = (*env)->NewStringUTF(env, "123"); /* 3. perform operation on the object */
(*env)->SetObjectField(env, obj, fid, jstr);
```

Calling a Java Method

1. Find the class of the object
   - Call GetObjectClass
2. Find the method ID of the object
   - Call GetMethodID, which performs a lookup for the Java method in a given class
3. Call the method
   - JNI provides an API for each type of method
     - e.g., CallVoidMethod(), etc.
   - You pass the object, method ID, and the actual arguments to the method (e.g., CallVoidMethod)

Example of Call:

```
jclass cls = (*env)->GetObjectClass(env, obj);
jmethodID mid = (*env)->GetMethodID(env, cls, "hello", "(I)V");
(*env)->CallVoidMethod(env, obj, mid, parm1);
```

Garbage Collection Issues

- Only Arrays and explicitly globally created objects are “pinned” down and must be explicitly released
- Everything else is released upon the native method returning

Thread Issues

- The JNI interface pointer (JNIEnv *) is only valid in the current thread
  - You must not pass the interface pointer from one thread to another
  - You must not pass local references from one thread to another
  - Check the use of global variables carefully (locking is needed)

Synchronization

- Synchronization is available as a C call
- Wait and Notify calls through JNIEnv do work and are safe to use
- Could use native threading operations for native to native threading, but this may cost portability

In java:

```
synchronized (obj)
{ ...
      /* synchronized block */
  ...
}
```

In C:

```
(*env)->MonitorEnter(env, obj);

/* synchronized block */

(*env)->MonitorExit(env, obj);
```

Embedding a VM in C

- Just a special kind of Java Call from C (see reference)
- You get a pointer into your resulting environment and by the VM are treated as a native method
- With the exception you never “return” so it is your responsibility to do everything globally

References

- http://docs.oracle.com/javase/7/docs/technotes/guides/jni/