CMPSC 461

Programming Language Concepts

Gang Tan
Computer Science and Engineering
Penn State University

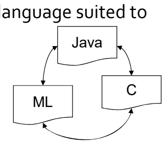
Language Interoperation

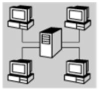
* Some slides are adapted from slides by John Mitchell

2

Why is Interoperability Important?

- Write each part of a complex system in a language suited to the task:
  - C for low-level machine management
  - Java/C#/Objective-C for user-interface
  - Ocaml/ML for tree transformations, parsers, …
- Integrate existing systems:
  - implemented in different languages
  - for different operating systems
  - on different underlying hardware systems



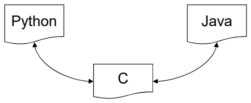


What's Involved?

- Languages make different choices:
  - Function calling conventions
    - caller vs callee saved registers
  - Data representations
    - strings, object layout
  - Memory management
    - tagging scheme
- Solution concepts
  - Stubs and wrappers
  - Data conversion
  - "Abstract"/opaque treatment of objects
    - Method calls go back to language where object was defined

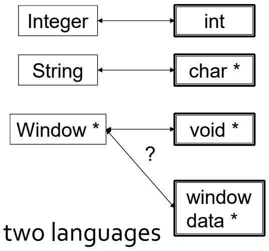
C/C++ as Lingua Franca

- Ubiquitous
- Computation model *is* underlying machine:
  - Other languages already understand
  - No garbage collection
- Representations well-known and fixed
  - Millions of lines of code would break if changed



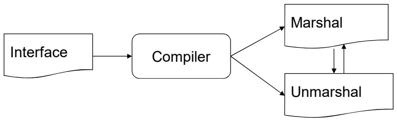
Marshaling and Unmarshaling

- Convert data representations from one language to another
- Easier when one end is C as rep is known
- Policy choice: copy or leave abstract?
- Tedious, low-level
- Modulo policy, fixed by two languages



Interface Specifications

- Contract describing what an implementation written in one language will provide for another
  - Inferred from high-level language: JNI
  - Inferred from C header files: SWIG
  - Specified in Interface Definition Language: ocamlidl, COM, CORBA
- Allow tools to generate marshalling/unmarshalling code automatically



Interface
Compiler
Marshal
Unmarshal

Foreign Function Interfaces (FFIs)

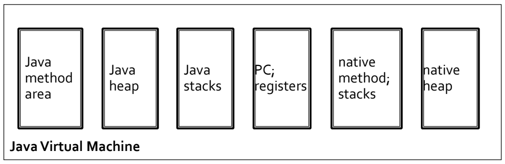
- Most languages provide an FFI
  - Java Native Interface
  - OCaml/C
  - Python/C
  - Haskell/C,
  - …

JNI: Integrating C/C++ and Java

- Java Native Interface
  - Allows Java methods to be implemented in C/C++
  - Such native methods can
    - create, inspect, and send messages to Java objects
    - modify Java objects
    - catch and throw exceptions in C that Java will handle
- JNI enforces policy: object references are abstract

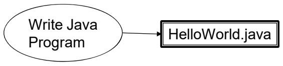
JVM memory areas

- Separate memory area for native methods
  - Pass data to native methods
    - Convert if primitive type
    - Pass pointer to Java heap otherwise



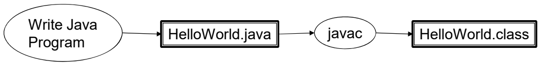
Java method area
Java heap
Java stacks
PC; registers
native method; stacks
native heap
Java Virtual Machine

JNI Example: Hello World!

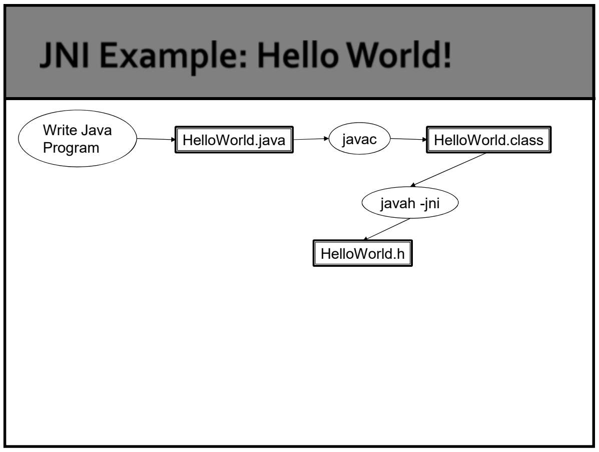


Write Java Program
HelloWorld.java

JNI Example: Hello World!

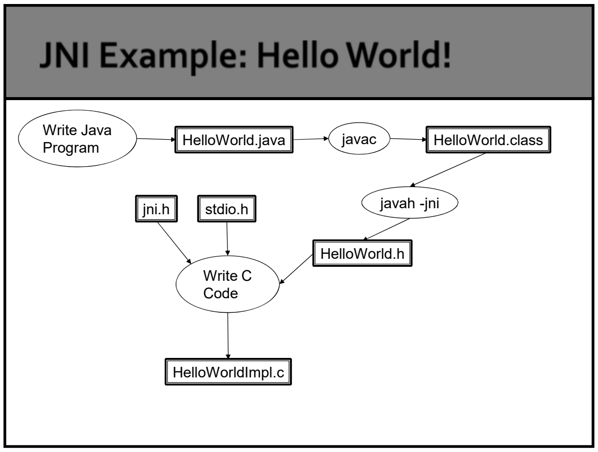


Write Java Program
HelloWorld.java
javac
HelloWorld.class



JNI Example: Hello World!

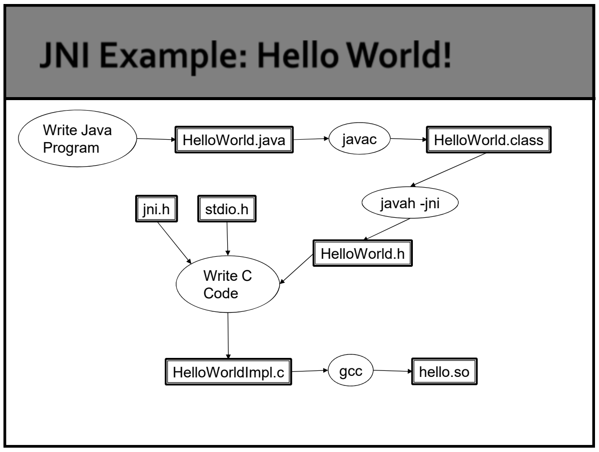
Write Java Program → HelloWorld.java → javac → HelloWorld.class → javah -jni → HelloWorld.h



JNI Example: Hello World!

Write Java Program → HelloWorld.java → javac → HelloWorld.class → javah -jni → HelloWorld.h

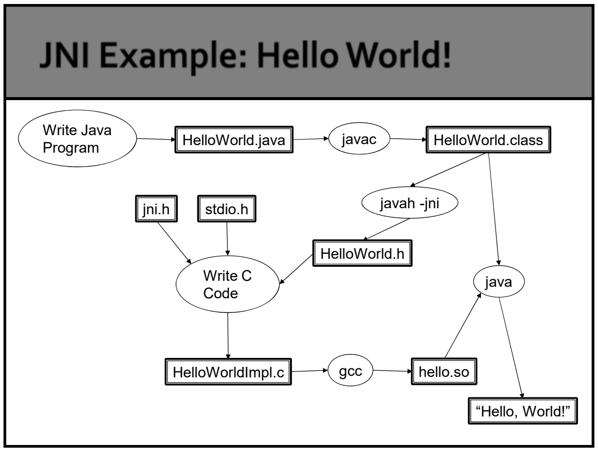
jni.h, stdio.h, HelloWorld.h → Write C Code → HelloWorldImpl.c



JNI Example: Hello World!

Write Java Program → HelloWorld.java → javac → HelloWorld.class → javah -jni → HelloWorld.h

jni.h, stdio.h, HelloWorld.h → Write C Code → HelloWorldImpl.c → gcc → hello.so

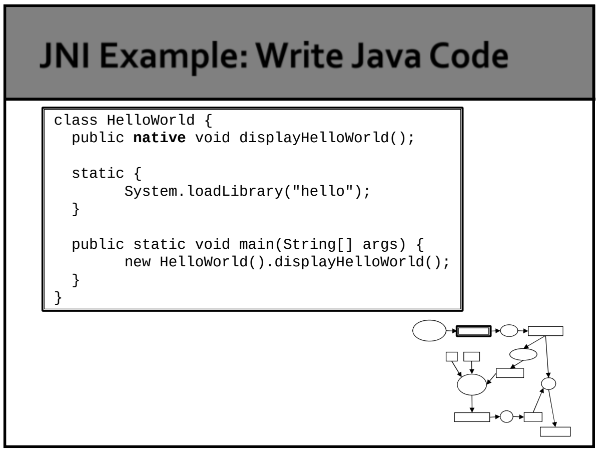


JNI Example: Hello World!

Write Java Program → HelloWorld.java → javac → HelloWorld.class → javah -jni → HelloWorld.h

jni.h, stdio.h, HelloWorld.h → Write C Code → HelloWorldImpl.c → gcc → hello.so

HelloWorld.class, hello.so → java → "Hello, World!"

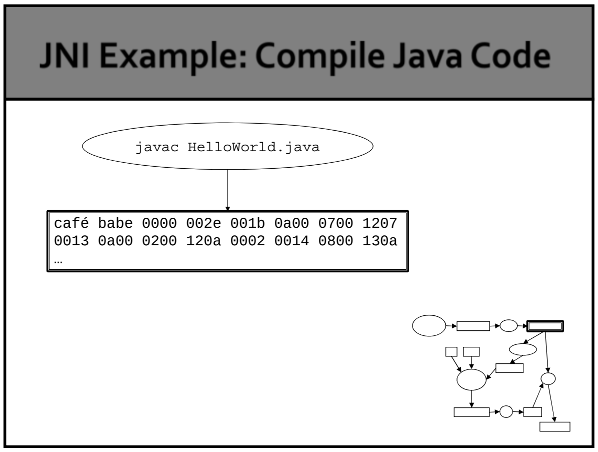


JNI Example: Write Java Code

```
class HelloWorld {
  public native void displayHelloWorld();

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

  public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
  }
}
```



JNI Example: Compile Java Code

```
javac HelloWorld.java
```

```
café babe 0000 002e 001b 0a00 0700 1207
0013 0a00 0200 120a 0002 0014 0800 130a
…
```

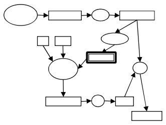
JNI Example: Generate C Header

```
javah -jni
HelloWorld
```

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

JNIEXPORT void JNICALL
  Java_HelloWorld_displayHelloWorld
  (JNIEnv *, jobject);
#endif
```

- Function has two "extra" args
  - Environment pointer
    - Provides access in C to JNI functions, e.g., function to convert Java string to char *
  - Object pointer (*this*)

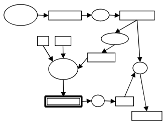


JNI Example: Write C Method

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

JNIEXPORT void JNICALL
Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj) {
    printf("Hello world!\n");
    return;
}
```

Implementation includes 3 header files:
- jni.h: provides information that C needs to interact with JVM
- HelloWorld.h: generated in previous step
- stdio.h: provides access to `printf`



JNI Example: Create Shared Lib

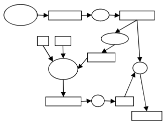
How to create a shared library depends on platform:

Cent OS (SunLab):

```
gcc -shared -fPIC -I/usr/java/latest/include
    -I/usr/java/latest/include/linux
    HelloWorld.c -o libhello.so
```

Microsoft Windows w/ Visual C++:

```
cl -Ic:\java\include
   -Ic:\java\include\win32
   -LD HelloWorldImp.c -Fehello.dll
```

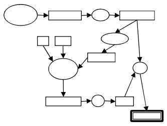


JNI Example: Run Program

```
java HelloWorld
```

```
Hello World!
```

Need to first set the library path to include the directory where libhello.so is; in linux, change LD_LIBRARY_PATH



JNI: Type Mapping

- Java primitive types map to corresponding types in C
- All Java object types are passed by reference (jobject)

Java type	Native C type	Description
bool	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
long	jint	signed 32 bits
long long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits

JNI: Method Mapping

- The **`javah`** tool uses type mapping to generate prototypes for native methods:

JNI: Accessing Java Strings

- Type **`jstring`** is not **`char *`**!
- Native code must treat **`jstring`** as an abstract type and use **`env`** functions to manipulate a jstring

Example of Handling Java strings

```
JNIEXPORT jstring JNICALL
Java_Prompt_getLine(JNIEnv *env, jobject obj, jstring prompt)
{
 char buf[128];
 const jbyte *str;
 str = (*env)->GetStringUTFChars(env, prompt, NULL);
 if (str == NULL) {
   return NULL; /* OutOfMemoryError already thrown */
 }
 printf("%s", str);
 (*env)->ReleaseStringUTFChars(env, prompt, str);
 /* We assume here that the user does not type more than
 * 127 characters */
 scanf("%s", buf);
 return (*env)->NewStringUTF(env, buf);
}
```

JNI: Calling Methods

- Native methods can invoke Java methods using the environment argument:

Method signature
(int) void

```
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);
}
```

Method name

Code uses `CallVoidMethod` because return type of callback method is void

Error handling

- Two difficult areas for interoperability
  - Memory management
  - Error handling
- JNI native methods can catch, throw exceptions

An Example For Exception Handling

```
JNIEXPORT void JNICALL
Java_CatchThrow_doit(JNIEnv *env, jobject obj)
{
 …
 (*env)->CallVoidMethod(env, obj, mid);
 exc = (*env)->ExceptionOccurred(env);
 if (exc) {
   /* We don't do much with the exception, except that
   we print a debug message for it, clear it, and
   throw a new exception. */
   jclass newExcCls;
   (*env)->ExceptionDescribe(env);
   (*env)->ExceptionClear(env);
   newExcCls = (*env)->FindClass(env,
                   "java/lang/IllegalArgumentException");
   if (newExcCls == NULL) {
   /* Unable to find the exception class, give up. */
     return;
   }
  (*env)->ThrowNew(env, newExcCls, "thrown from C code");
 }
}
```

JNI: Summary

- Allows Java methods to be implemented in C/C++
- Interface determined by native method signature
- Tools generate C interfaces and marshaling code
- References are treated abstractly, which facilitates memory management
- Environment pointer provides access to JVM services such as object creation and method invocation

- References
  - **The Java Native Interface:** Programmer's Guide and Specification
    - http://www.worldcolleges.info/sites/default/files/jni.pdf
  - API reference: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

SWIG

- Tool to make C/C++ libraries easily available in many high level languages:

  Tcl, Python, Perl, Guile, Java, Ruby, Mzscheme, PHP, Ocaml, Pike, C#, Allegro CL, Modula-3, Lua, Common Lisp, JavaScript, Eiffel, …

- Goal: Infer interface from C/C++ headers, requiring annotations only to customize.
- Marshaling policy: references treated opaquely. C library must provide extra functions to allow high-level language to manipulate.

www.swig.org