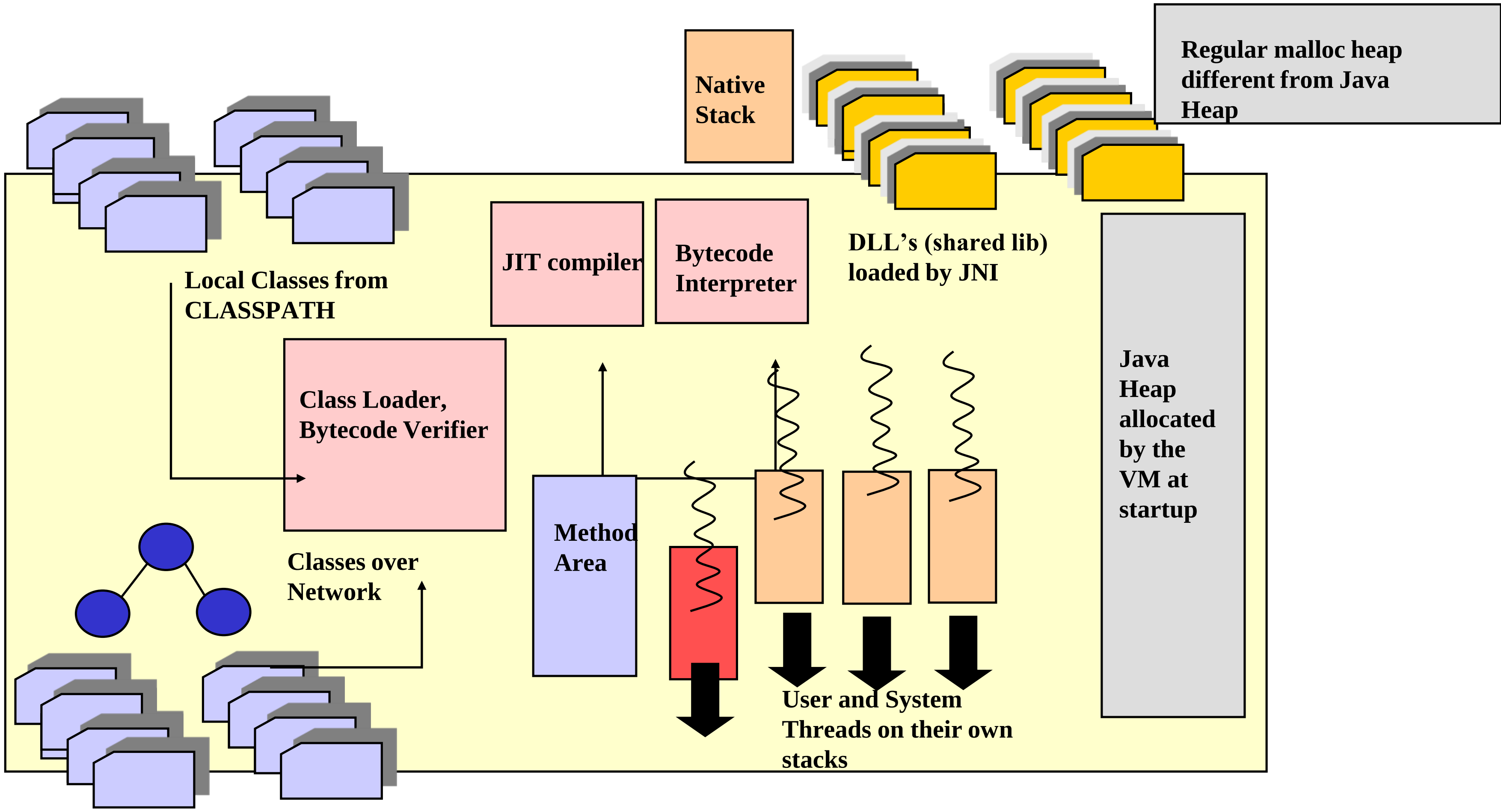
Java Native Interface

Bob Singh

Introduction

- Java Native Interface (JNI) is a standard programming interface for writing Java native methods and embedding the JVM into native applications.
- The primary goal is binary compatibility of native method libraries across all Java virtual machine implementations on a given platform.

Java Runtime Environment

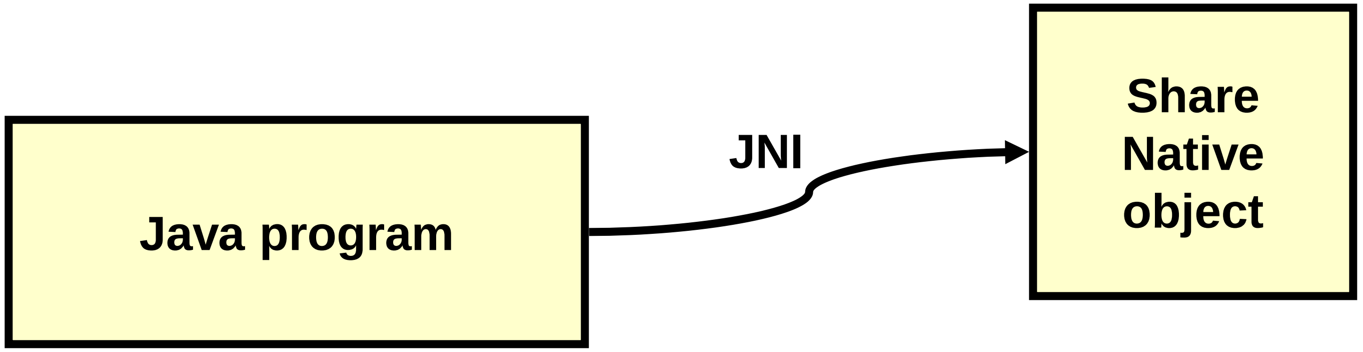

Native Stack
Regular malloc heap different from Java Heap
JIT compiler
Bytecode Interpreter
DLL's (shared lib) loaded by JNI
Local Classes from CLASSPATH
Class Loader, Bytecode Verifier
Java Heap allocated by the VM at startup
Method Area
Classes over Network
User and System Threads on their own stacks

Native Methods--What?

- Through Java Native Interface (JNI), Java supports the ability to call native methods--shared object files written in languages such as C/C++

- The native keyword is used to identify methods that have implementation defined in such shared object files:

```
public void native sayHello();`
```



Native Methods -- Why?

- Before writing JNI, consider:
  - Could the program be rewritten in/ported to Java?
    - It’s possible to write programs quickly in Java
  - What happens to portability?

- JNI provides a means of providing cross-platform native code, but only if appropriate libraries are generated for each machine. (Code is portable, but not WORA--write once, run anywhere.)

Step 1: Organizing JNI vs. Java Functionality

First, decide what code belongs with native methods, and what is better left to Java. Consider:

1) **JNI means loss of visibility modifiers (even private members can be discovered!)**

2) **JNI means manual garbage collection-- difficult at times**

3) **JNI can imply a loss of OO control, unless native implementation is in C++, or a very OO-oriented set of C libraries.**

```
public class HelloWorld
{

   public native void sayHello(String strMessage);


   public void speakUp()
   {

      System.out.println
            ("Java Says Hello, C");
      sayHello("C Says Hello, Java");

   }// speakUp()


}// class HelloWorld
```

Step 2: Generate Header File

1) Use javah tool from JDK to create header file.

2) Target your compiled class file.

3) Be sure to use the -jni switch:

```
%javah -jni HelloWorld
```

4) For older, non-JNI header files, use the -stub option with javah. (Ask: why are you using the old native invocation anyway? Consider upgrading to 1.1)

5) Don’t edit the source Java file (e.g., even adding a package statements invalidates the header file.)

This should create a header file essential for your implementation . . .

```
/* DO NOT EDIT THIS FILE - it is machine generated */
       #include <jni.h>
       /* Header for class HelloWorld */

       #ifndef _Included_HelloWorld
       #define _Included_HelloWorld
       #ifdef __cplusplus
       extern "C" {
       #endif
       /*
        * Class:     HelloWorld
        * Method:    sayHello
        * Signature: (Ljava/lang/String;)V
        */
        JNIEXPORT void JNICALL Java_HelloWorld_sayHello
          (JNIEnv *, jobject, jstring);

        #ifdef __cplusplus
        }
        #endif
        #endif
```

See the "Do Not Edit" Warning? Obey!

Step 3: Implement Your JNI Method

1) #include the header from javah

2) Use the function prototypes generated by javah

3) Remember that Java Strings are objects, not char arrays.

4) Consult the JNI API for a list of helpful functions and methods

5) Note that ALL functions have two parameters, at least:

```
JNIEnv * env, jobject thisObj
```

These are references to the VM and “this” object, respectively. They are the window into the process running in the VM.

```
 #include <stdio.h>
 #include "HelloWorld.h"
 JNIEXPORT void JNICALL

 Java_HelloWorld_sayHello
   (JNIEnv * env, jobject thisObj, jstring strMessage){

  const char *str =
       (*env)->GetStringUTFChars(env, strMessage, 0);

  printf("%s\n", str);

  (*env)->ReleaseStringUTFChars(env, strMessage, str);

}// JNI method

 /*Note need to convert chars from Unicode to UTF,
 and then free created char buffer from the VM */
```

4. Compile the Object

Observe the naming convention:

```
                    lib<ShareObject>.so
```

LINUX:

```
  %cc -I$JDK_HOME/include -I$JDK_HOME/include/genunix \
           -shared -o libHello.so Hello.c
```

SOLARIS:

```
  %cc  -I$JDK_HOME/include -I$JDK_HOME/include/solaris \
           -G -o libHello.so Hello.c
```

WINDOWS:

Consider using a GNU port, or the upcoming batch file ...

```
ftp://go.cygnus.com/pub/ftp.cygnus.com/
             gnu-win32/gnu-win32-b18/cdk.exe
```

```
#
# Makefile for Linux JNI Compilation
#
UPATH    = /usr/bin/
JDK_PATH     = /usr/local/jdk117_v1a/
# define utility programs and options
CC   = $(UPATH)cc
CFLAGS   = -I$(JDK_PATH)include -I$(JDK_PATH)include/genunix -shared
MAKE     = $(UPATH)make
CTAGS    = $(UPATH)ctags
INDENT   = $(UPATH)indent -bl -c41 -i4 -l72 -pcs
# default target - builds Hello executable
#
Hello:  Hello.c
    $(CC) $(CFLAGS) -o libHello.so Hello.c

# "debug" target - builds executable with debug code
#
debug:  Hello.c
    @CFLAGS="$(CFLAGS) -DDEBUG";export CFLAGS;$(MAKE) -e

# "pretty" target - beautify source files
pretty: Hello.c
    ls $? | xargs -p -n1 $(INDENT)
    @touch pretty
# "clean" target - remove unwanted object files and executables
clean:
    rm -f Hello Hello.o pretty tags lint nohup.out a.out core
```

For Solaris, change to /solaris and -G

Windows Batch File

```
@echo Batch File for JNI W95 DevStudio 5
@echo Making clean
@del *.obj
@del *.lib
@del *.dll
@del *.pch
@del *.pdb
@del *.exp
@del *.idb
@echo Compiling all C files from this directory...
@cl -I%JDKPATH%\include -I%JDKPATH%\include\win32
       /nologo /MTd /W4 /Gm /GX /Zi /Od /D "WIN32"
       /D "_DEBUG" /D "_WINDOWS" /YX  /c /LD *.c

@echo linking ...
@link /nologo /subsystem:windows /dll /incremental:no
      /machine:I386 /out:%1.dll *.obj
```

5. Load the Library

```
public class Test{

static {
  /*
   *  Our library is in a file called "libHello.so", but
   *  just pass in "Hello" since Java will prepend "lib"
   *  and append the ".so" extension.  Windows users
   *  should omit the “.dll” extension as well.
   */
  System.loadLibrary("Hello");
  }// static load

 public static void main (String arg[]) {
     HelloWorld hw = new HelloWorld();
     hw.speakUp();

 } // main

}// class Test
```

6. Execute

1) Make sure the library is in the path of the environment variable:

```
LD_LIBRARY_PATH
```

2) For your shell, you can set:

```
LD_LIBRARY_PATH=./:$LD_LIBRARY_PATH
```

3) Windows users need to set the PATH appropriately:

```
PATH=%PATH%;c:\MyLibrary
```

7. DEBUG (and repeat . . .)

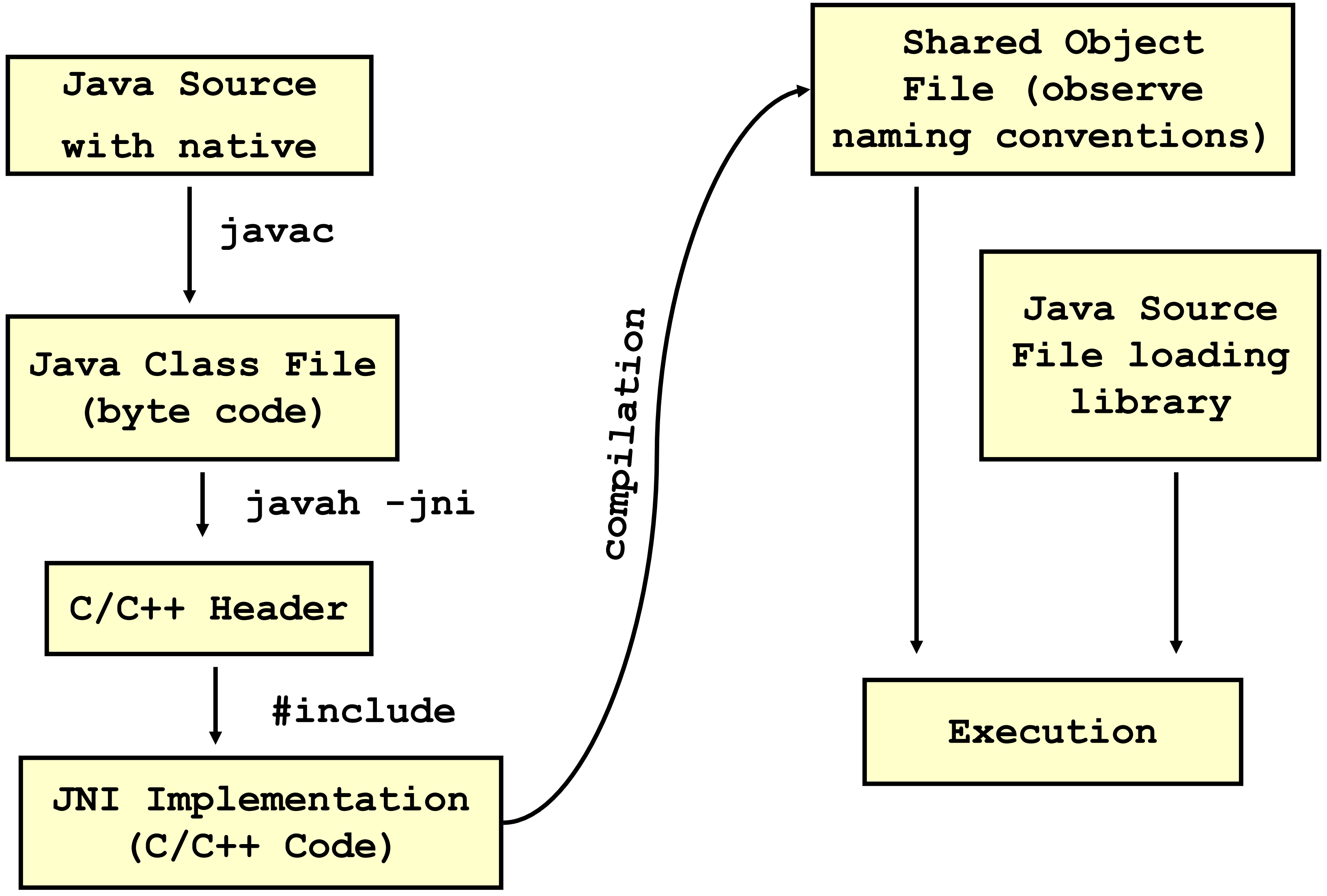
Numerous problems may come up during compilation.

Resources for debugging JNI problems:

`http://www.codeguru.com/java/JNI/index.shtml`

`http://www.mindspring.com/~david.dagon/jni/Native.txt`

JNI Generation--Overview


Java Source with native
javac
Java Class File (byte code)
javah -jni
C/C++ Header
#include
JNI Implementation (C/C++ Code)
compilation
Shared Object File (observe naming conventions)
Java Source File loading library
Execution