Java Native Interface

JNI is a mechanism that allows

- a Java program to call a function in a C or C++ program.
- a C or C++ program to call a method in a Java program.

Reasons for using Native Methods

- Get access to system features that can be handled more easily in C or C++.
- Need access to legacy code that has been well tested.
- Need the performance that C provides and that Java does not have (yet).

What about Languages other than C and C++?

- Tools for working with other languages can be built, but have not been yet.
- Alternative: Go through C or C++ to get to other programming languages.

Important: Native code is *not* portable

- The Java part is portable.
- Everything else is machine dependent.

Steps In Using JNI

1. Java code (write and compile)

   - Declare native methods using **native** and no body.

     public native void nativeOne();

   - Ensure that a shared library, to be created later, is loaded before the native method is called.

     System.loadLibrary("NativeLib");

     Usually executed in a static initializer block in the class that calls the native method(s).

2. Create a C header file containing function prototypes for the native methods.

   javah -jni NativeMethods

   where NativeMethods is the Java class containing the native methods.

3. Write C implementations of native methods using mangled names and extra parameters.

   Native (C/C++) code function names are formed from the following pieces:

   'Java_'

   Mangled fully qualified class name with periods becoming underscores

   '_'

   Mangled method name

For overloaded Java methods:

Continue with two underscores '__'

Mangled argument types

Better: Get prototype from header file created in step 2.

Linux and HP Version

```
JNIEXPORT
    void JNICALL
        Java_NativeMethods_nativeOne(
                JNIEnv * env, jobject thisObj)
```

Note: JNIEXPORT and JNICALL are currently defined to be nothing.

4. Compile C code and Create shared library

```
% gcc -I/usr/java/j2sdk1.4.2_05/include
    - I /usr/java/j2sdk1.4.2_05/include/linux
        -shared cImplOne.c cImplTwo.c
            -o libNativeLib.so
```

Note that the name of the shared library has the prefix "lib" in front of it.

5. Execute Java program

```
% java Main
```

Example

A Java program with two native methods that perform simple output in C.

The Java main method and the native methods are in separate classes.

Following the code is a unix script file, called *linuxC*, that performs the steps to compile and execute the native code example.

```
// File: NativeMethods.java

public class NativeMethods
{
	public native void nativeOne();
	public native void nativeTwo();
}
```

```
// File: Main.java

// Loads a native library.
// Creates an object and invokes native methods.

public class Main
{
	static
	{
		System.loadLibrary("NativeLib");
	}

	public static void main(String [] args)
	{
		NativeMethods nm = new NativeMethods();
		nm.nativeOne();
		nm.nativeTwo();
	}
}
```

NativeMethods.h

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

#ifndef _Included_NativeMethods
#define _Included_NativeMethods
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     NativeMethods
 * Method:    nativeOne
 * Signature: ()V
 */
JNIEXPORT void JNICALL
            Java_NativeMethods_nativeOne
                                                (JNIEnv *, jobject);

/*
 * Class:     NativeMethods
 * Method:    nativeTwo
 * Signature: ()V
 */
JNIEXPORT void JNICALL
            Java_NativeMethods_nativeTwo
                                                (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
#endif
```

```
/* File: cImplOne.c

 * Implements nativeOne in NativeMethods
 */

#include <stdio.h>
#include "NativeMethods.h"

JNIEXPORT void JNICALL
        Java_NativeMethods_nativeOne
                (JNIEnv* env, jobject thisObj)
{
    printf("Hello Advanced Java World\n");
}


//-------------------------------------------------


/* File: cImplTwo.c

 * Implements nativeTwo in NativeMethods
 */

#include <stdio.h>
#include "NativeMethods.h"

JNIEXPORT void JNICALL
        Java_NativeMethods_nativeTwo
                (JNIEnv* env, jobject thisObj)
{
    printf("Hello from the second method\n");
}
```


File: linuxC

```
JNI_INCLUDE= "-I/usr/java/j2sdk1.4.2_05/include
                    -I/usr/java/j2sdk1.4.2_05/include/linux"
export LD_LIBRARY_PATH=.
echo "============================================="
echo
javac NativeMethods.java Main.java
echo 'Java files compiled'
echo
javah -jni NativeMethods
echo 'Header file created'
echo
gcc  $JNI_INCLUDE -shared cImplOne.c cImplTwo.c
                                        -o libNativeLib.so
echo 'Shared library created'
echo "============================================="
echo
echo ">>> Output <<<<"
java  Main
echo "============================================="
echo
```

Output

```
=====================================
Java files compiled
Header file created
Shared library created
=====================================
>>> Output <<<<
Hello Advanced Java World
Hello from the second method
=====================================
```

```
% ls
Main.java                   Main.class
NativeMethods.java          NativeMethods.class
                            NativeMethods.h
cImplOne.c                  cImplOne.o
cImplTwo.c                  cImplTwo.o
linuxC                      libNativeLib.so
```

Type Encoding (Mangling)

Basic codes

boolean	Z
char	C
byte	B
short	S
int	I
long	J
float	F
double	D
void	V
array of *type*	**[***type*
class name	**L***fully-qualified-name***;**
method type	**(***arg-types***)***result-type*

Name munging

.	→	_
/	→	_
unicode char	→	_0xxxx
_	→	_1
;	→	_2
[	→	_3

Sample Signatures and Prototypes

These methods are found in a class called Sigs.

public static native void staticMeth();

Signature: ()V

```
JNIEXPORT void JNICALL
        Java_Sigs_staticMeth(JNIEnv *, jclass);
```

public native void fun_meth();

Signature: ()V

```
JNIEXPORT void JNICALL
        Java_Sigs_fun_1meth__(JNIEnv *, jobject);
```

public native int fun_meth(**double** d, String s);

Signature: (DLjava/lang/String;)I

```
JNIEXPORT jint JNICALL
    Java_Sigs_fun_1meth__DLjava_lang_String_2
                (JNIEnv *, jobject, jdouble, jstring);
```

public native String fun_meth(Date [] d);

Signature: ([Ljava/util/Date;)Ljava/lang/String;

```
JNIEXPORT jstring JNICALL
    Java_Sigs_fun_1meth___3Ljava_util_Date_2
                (JNIEnv *, jobject, jobjectArray);
```

public native long [] fun_meth(**short** s, **byte** b, **char** c);

Signature: (SBC)[J

```
JNIEXPORT jlongArray JNICALL
        Java_Sigs_fun_1meth__SBC
            (JNIEnv *, jobject, jshort, jbyte, jchar);
```

Native Types Corresponding to Java Types

Java	C/C++
boolean	jboolean
char	jchar
byte	jbyte
short	jshort
int	jint
long	jlong
float	jfloat
double	jdouble
void	void
Object	jobject
Class	jclass
String	jstring
array	jarray
Throwable	jthrowable
true	JNI_TRUE
false	JNI_FALSE
	jsize

The definitions of these C/C++ types are machine dependent.

Every native method gets two additional parameters when translated into C or C++:

1. JNI Interface (environment) pointer

   JNIEnv * env

2. Object or class parameter

   For instance methods: jobject thisObj

   For class methods: jclass theClass

In the next example, note the conversion of Java types into the new C types and the additional parameters in the implementation of the native method.

Example

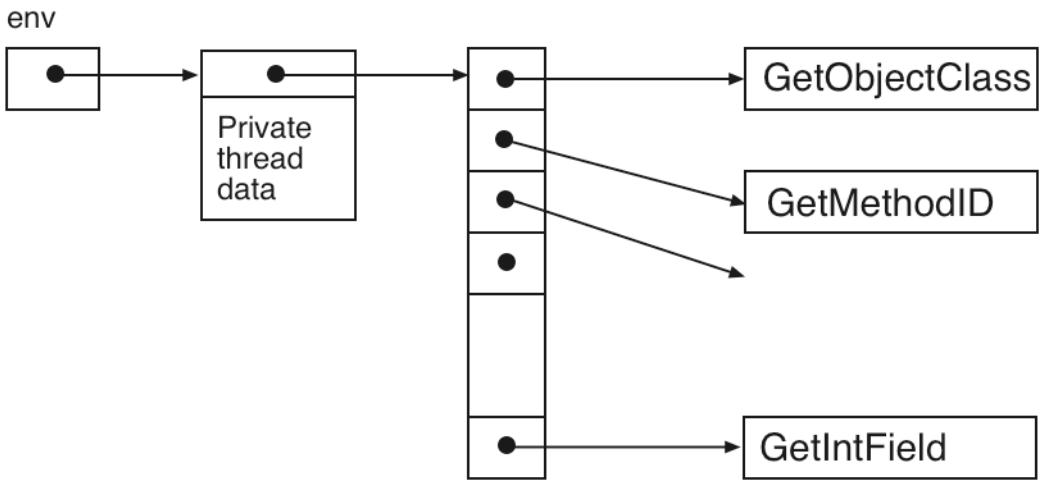
```
public class Compute
{
      static native double compute(int m, long n);

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

      public static void main(String [] args)
      {
            double answer = compute(45, 67L);

            System.out.println("answer = " + answer);
      }
}
```

```
#include "Compute.h"

JNIEXPORT jdouble JNICALL
        Java_Compute_compute__IJ
                (JNIEnv * env, jclass jClass, jint m, jlong n)
{
                /* assumes method is overloaded */
        jdouble result = m + n + 100;
        return result;
}
```

JNI Support

Over 200 native functions, declared in jni.h, can be used to support Java data types, objects, and classes in C and C++.

Every native method receives a JNIEnv pointer as its first parameter; this pointer provides access to the JNI support functions.



Example

Native function syntax

```
jclass GetObjectClass(JNIEnv * env, jobject ob)
```

Call in C

```
jclass c = (*env) -> GetObjectClass(env, obj);
```

Call in C++

```
jclass c = env -> GetObjectClass(obj);
```

Remember

```
(*env) -> GetObjectClass(env,obj)
```

is an abbreviation for

```
(*(*env)).GetObjectClass(env,obj)
```

Observe that C++ calls are simpler because its version of the JNIEnv class provides inline member functions that handle the function pointer lookup.

The environment pointer can be omitted from the parameter list when using C++.

A C++ Example

Use JNI functions to convert between a Java String and a C/C++ string in a function that reverses the string.

Write the native code in C++.

Java Class

```
public class Reverse          // File: Reverse.java
{
      static
      {    System.loadLibrary("Reverse"); }

      public static native String reverse(String s);

      public static void main(String [] arg)
      {
            String s = reverse("to be or not to be");
            System.out.println(s);
      }
}
```

Conversion functions (using UTF-8 strings)

```
const char * GetStringUTFChars(
            JNIEnv * env, jstring s, jboolean * isCopy)

jstring NewStringUTF(JNIEnv * env,
                                    const char * bytes)
```

C++ Code

```
#include "Reverse.h"            // File: reverse.cpp
#include <stdlib.h>
#include <string.h>

JNIEXPORT jstring JNICALL
                    Java_Reverse_reverse(
                        JNIEnv * env, jclass cl, jstring s)
{
    const char * inString =
                    env->GetStringUTFChars(s, NULL);

    int len = env->GetStringUTFLength(s);

    char * outString = (char *)malloc(strlen(inString)+1);

    for (int k=0; k<len; k++)
        outString[k] = inString[len-k-1];
    outString[len] = '\0';

    return env->NewStringUTF(outString);
}
```

We need a slightly different unix script file, called *linuxCPP*, that performs the steps to compile and execute the native code example using the C++ implementation.

File: linuxCPP

```
JNI_INCLUDE= "-I/usr/java/j2sdk1.4.2_05/include
                    -I/usr/java/j2sdk1.4.2_05/include/linux"
export LD_LIBRARY_PATH=.
echo "==========================================="
echo
javac Reverse.java
echo 'Java files compiled'
echo
javah -jni Reverse
echo 'Header file created'
echo
g++  -Wall $JNI_INCLUDE -shared reverse.cpp
                                          -o libReverse.so
echo 'Shared library created'
echo "==========================================="
echo
echo ">>> Output <<<<"
java  Reverse
echo "==========================================="
echo
```

Output

```
=====================================
Java files compiled
Header file created
Shared library created
=====================================
>>> Output <<<
eb ot ton ro eb ot
=====================================
```

Two Other Types

jfieldID	for variable names
jmethodID	for method names

Altering Variables

To access or change an instance variable or a class variable, we need this information:

- Class object of type jclass for the class containing the variable.
- Name of the field as a C string.
- Type (signature) of the field as a C string.

To get a field object of type jfieldID use:

```
jfieldID GetFieldID(
        JNIEnv * env, jclass cl,
                const char * name, const char * sig)
```

JNI has two different sets of functions for accessing and setting variables, one for class variables and one for instance variables.

Each function includes the type of the datum being accessed in its identifier.

```
jint GetIntField(JNIEnv * env, jobject ob, jfieldID name)

void SetIntField(JNIEnv * env, jobject ob,
                                jfieldID name, jint val)
```

Accessor and Mutation Functions

<jniType> Get*<fType>*Field(JNIEnv * env,
jobject ob, jfieldID name)

<jniType> GetStatic*<fType>*Field(JNIEnv * env,
jclass c, jfieldID name)

void Set< *fType*>Field(JNIEnv * env,
jobject ob, jfieldID name, <jniType> val)

void SetStatic< *fType*>Field(JNIEnv * env,
jclass c, jfieldID name, <jniType> val)

where *<fType>* can be one of the following:

Boolean, Char, Byte, Short,

Int, Long, Float, Double, Object.

and *<jniType>* can be one of the following:

jboolean, jchar, jbyte, jshort,

jint, jlong, jfloat, jdouble, jobject.

Consequence

Private and protected variables can be accessed and altered from outside of their class by means of native functions.

Example

SomeVars: A class with two private variables.

SetSomeVars: A class with two native methods that will alter the private variables in a SomeVars object.

Driver: A class with a main method that creates a SomeVars object and a SetSomeVars object and modifies the private variables by calling the native methods.

File: SomeVars.java

```
public class SomeVars
{
    private int aPrivate = 5;
    private static double aStaticPrivate = 7.77;

    public void printAPrivate()
    {
        System.out.println("aPrivate = " + aPrivate);
    }

    public void printAStaticPrivate()
    {
        System.out.println("aStaticPrivate = " + aStaticPrivate);
    }
}
```

File: SetSomeVars.java

```
public class SetSomeVars
{
    public native void setAPrivate(SomeVars v, int val);
    public native void setAStaticPrivate(Class c, double val);
}
```

File: Driver.java

```
public class Driver
{
    static
    {   System.loadLibrary("SomeVars");   }
```

```
    public static void main(String [] args)
                                throws ClassNotFoundException
    {   SomeVars sv = new SomeVars();
        SetSomeVars ssv = new SetSomeVars();

        sv.printAPrivate();
        sv.printAStaticPrivate();

        ssv.setAPrivate(sv, 135);
        Class svClass = Class.forName("SomeVars");
//      OR:   Class svClass = sv.getClass();
        ssv.setAStaticPrivate(svClass, 246.8);

        sv.printAPrivate();
        sv.printAStaticPrivate();
    }
}
```

File: setvars.cpp

```
#include "SetSomeVars.h"

JNIEXPORT void JNICALL
        Java_SetSomeVars_setAPrivate(
                JNIEnv *env, jobject thisObj,
                                        jobject obj, jint val)
{
    jclass c = env->GetObjectClass(obj);
    jfieldID fid = env->GetFieldID(c, "aPrivate", "I");
    env->SetIntField(obj, fid, val);
}

JNIEXPORT void JNICALL
        Java_SetSomeVars_setAStaticPrivate(
                JNIEnv * env, jobject thisObj, jclass c, jdouble val)
{
```



```
    jfieldID fid = env->GetStaticFieldID(c, "aStaticPrivate", "D");
    env->SetStaticDoubleField(c, fid, val);
}
```

Output

```
% linuxCPP
=======================================
Java files compiled
Header file created
Shared library created
=======================================
>>> Output <<<
aPrivate = 5
aStaticPrivate = 7.77
aPrivate = 135
aStaticPrivate = 246.8
=======================================
```

Calling Java Methods

Methods can be called from native code, even private methods.

Follow these steps

- Get the class of the object that contains methods to be called.
  jclass GetObjectClass(JNIEnv * env, jobject ob)

- Get a method ID "object" for the method to be called using the class, the method name, and the method signature as parameters.

jmethodID GetMethodID(JNIEnv * env, jclass cl,
const char * name, **const char** * sig)

- Call the method using the object (for instance methods) or the class (for class methods) that owns it, the method ID object, and the arguments to the method as parameters.

jdouble CallDoubleMethod(jobject obj, jmethodID m, jint n)

Functions for Calling Methods

Call<*type*>Method(JNIEnv * env, jobject o, jmethodID m, …)

CallStatic<*type*>Method(JNIEnv * env, jclass c,
jmethodID m, …)

where <*type*> can be one of the following:

Void, Boolean, Char, Byte, Short,
Int, Long, Float, Double, Object,

and "…" represents the arguments passed to the method.

Note: JNI has two other sets of method for calling functions that handle the arguments differently as well as a set for "nonvirtual" methods.

- One set takes an array of *jvalue* for parameters.
- The other set uses the *va_list* mechanism.

These functions allow native methods to violate the security principles of Java:

Private and protected methods can be invoked.

Example

Call a private instance method and a private class method from outside of their class.

File: SomeMethods.java

```
public class SomeMethods
{
      private double methOne(int m, double x)
      {
            return m + x;
      }

      private static void methTwo(String s)
      {
            System.out.println(s);
      }
}
```

File: CallMethods.java

```
public class CallMethods
{
  public native double callOne(SomeMethods s);
  public native void callTwo(Class c);
}
```

File: MainCaller.java

```
public class MainCaller
{
      static
      {   System.loadLibrary("SomeMethods");   }
```

```
    public static void main(String [] args)
                                throws ClassNotFoundException
    {
        SomeMethods sm = new SomeMethods();
        CallMethods cm = new CallMethods();
        double d = cm.callOne(sm);
        System.out.println("d = " + d);
        Class smClass = Class.forName("SomeMethods");
        cm.callTwo(smClass);
    }
}
```

File: methods.cpp

```
#include "CallMethods.h"

JNIEXPORT jdouble JNICALL
        Java_CallMethods_callOne
            (JNIEnv * env, jobject thisObj, jobject obj)
{
  jclass cl = env->GetObjectClass(obj);
  jmethodID mid =
            env->GetMethodID(cl, "methOne", "(ID)D");
  return env->CallDoubleMethod(obj, mid, -99, -6.6);
}

JNIEXPORT void JNICALL
        Java_CallMethods_callTwo
            (JNIEnv *env, jobject thisObj, jclass cl)
{
  jmethodID mid = env->GetStaticMethodID(cl, "methTwo",
                                "(Ljava/lang/String;)V");
```

```
    jstring js = env->NewStringUTF(
                        "Pack my box with five dozen liquor jugs.");
    env->CallStaticVoidMethod(cl, mid, js);
}
```

Output

```
% linuxCPP
========================================
Java files compiled
Header file created
Shared library created
========================================
>>> Output <<<
d = -105.6
Pack my box with five dozen liquor jugs.
========================================
```

Obtaining Signatures

The utility operation *javap* can be used to identify the signatures of the methods in a class.

```
% javap -s -private SomeMethods
Compiled from SomeMethods.java
public synchronized class SomeMethods
                                    extends java.lang.Object
      /* ACC_SUPER bit set */
{
      private double methOne(int, double);
            /*   (ID)D   */
```

```
    private static void methTwo(java.lang.String);
        /*  (Ljava/lang/String;)V  */
    public SomeMethods();
        /*  ()V  */
}

% javap -s -private SomeVars
Compiled from SomeVars.java
public synchronized class SomeVars
                        extends java.lang.Object
/* ACC_SUPER bit set */
{
    private int aPrivate;
        /*  I  */

    private static double aStaticPrivate;
        /*  D  */

    public void printAPrivate();
        /*  ()V  */

    public void printAStaticPrivate();
        /*  ()V  */

    public SomeVars();
        /*  ()V  */

    static static {};       // To initialize class variables
        /*  ()V  */
}
```

Using javap

% javap -help

Usage: javap <options> <classes>...

where options include:

-b Backward compatibility with javap in JDK 1.1

-c Disassemble the code

-classpath <directories separated by colons>
List directories in which to look for classes

-extdirs <dirs>
Override location of installed extensions

-help Print this usage message

-J<flag> Pass <flag> directly to the runtime system

-l Print line number and local variable tables

-public Show only public classes and members

-protected Show protected/public classes and members

-package Show package/protected/public classes and members (default)

-private Show all classes and members

-s Print internal type signatures

-bootclasspath <pathlist>
Override location of class files loaded by bootstrap class loader

-verbose Print stack size, number of locals and args for methods

Constructing Objects in Native Code

Instances of Java classes can be constructed and used in native code functions using the following steps.

- Get a class object from a string name.

  jclass FindClass(JNIEnv * env, **const char** * className)

- Get a constructor method ID object using the signature of the constructor. Note <init> string.

  jmethodID GetMethodID(JNIEnv * env, jclass c,
  "<init>", **const char** * signature)

- Create the new object, passing actual parameters to the constructor.

  jobject NewObject(JNIEnv * env, jclass c, jmethodID m, …)

Note: JNI has two other object creation functions that handle the constructor parameters differently.

Example

Construct Domino objects in native code.

File: MakeDominoes.java

```
public class MakeDominoes
{
   static
   {   System.loadLibrary("Dominoes");   }

   public static native Domino createDomino();
```

```
   public static native Domino createDomino(
                                          int m, int n, boolean b);
   public static native Domino createDomino(boolean b);

   public static void main(String [] args)
   {
      Domino d1 = (Domino)createDomino();
      Domino d2 = (Domino)createDomino(2, 6, true);
      Domino d3 = (Domino)createDomino(false);
      Object d  = createDomino(1, 9, true);

      System.out.println("d1 = " + d1);
      System.out.println("d2 = " + d2);
      System.out.println("d3 = " + d3);
      System.out.println("d  = " + d);
   }
}

class Domino
{
   private int spots1, spots2;
   private boolean faceUp;
      :         :
}
```

File: dominoes.cpp

```
#include "MakeDominoes.h"

JNIEXPORT jobject JNICALL
            Java_MakeDominoes_createDomino__
                              (JNIEnv * env, jclass thisClass)
{
```

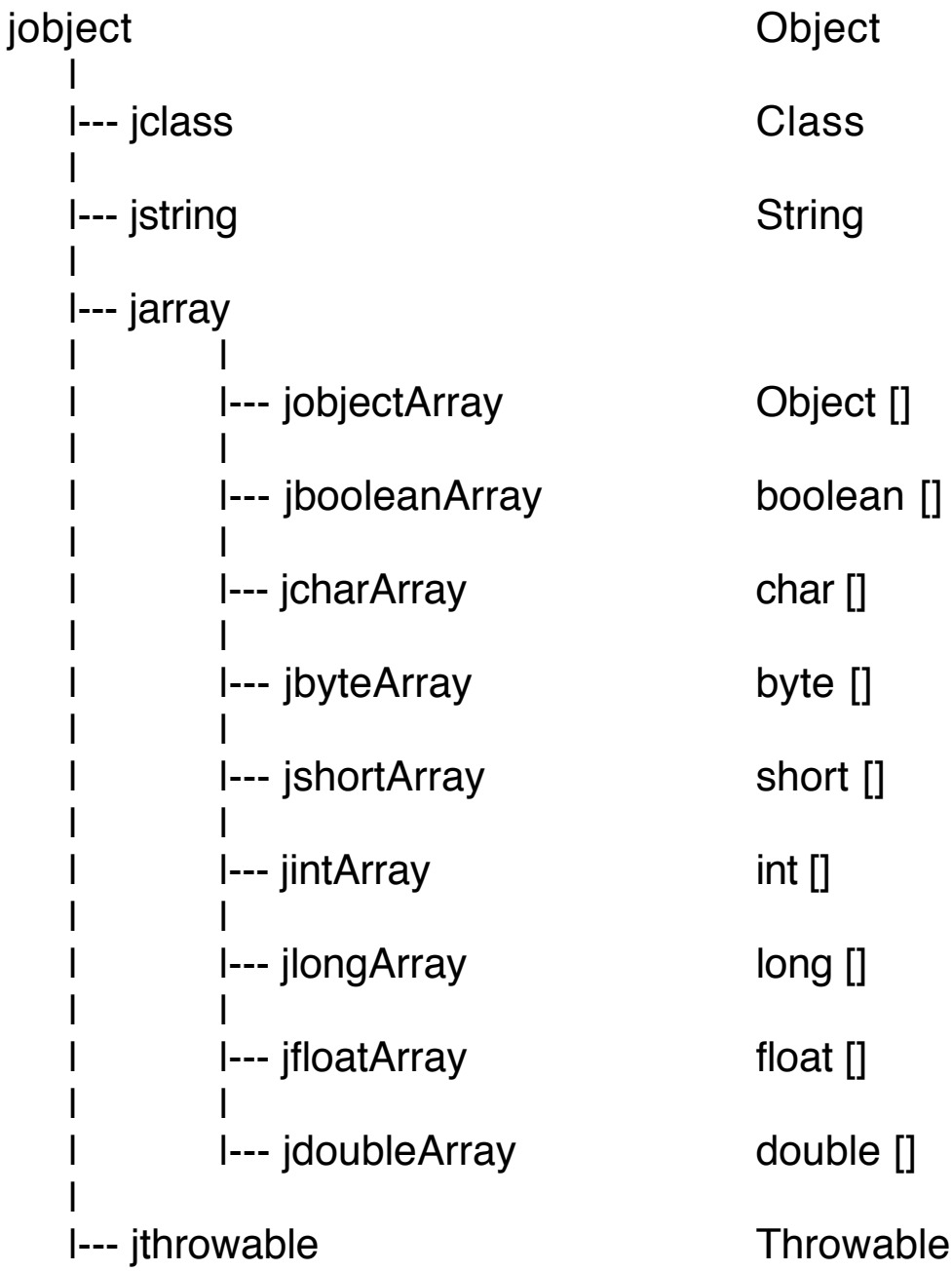
```
    jclass c = env->FindClass("Domino");
    jmethodID cid = env->GetMethodID(c, "<init>", "()V");
    jobject newObj = env->NewObject(c, cid);
    return newObj;
}

JNIEXPORT jobject JNICALL
        Java_MakeDominoes_createDomino__IIZ
            (JNIEnv * env, jclass thisClass,
                        jint m, jint n, jboolean b)
{
    jclass c = env->FindClass("Domino");
    jmethodID cid = env->GetMethodID(c, "<init>", "(IIZ)V");
    return env->NewObject(c, cid, m, n, b);
}

JNIEXPORT jobject JNICALL
        Java_MakeDominoes_createDomino__Z
           (JNIEnv * env, jclass thisClass, jboolean b)
{
    jclass c = env->FindClass("Domino");
    jmethodID cid = env->GetMethodID(c, "<init>", "(Z)V");
    return env->NewObject(c, cid, b);
}
```

The native methods return entities of type jobject. These come back to Java as type Object and must be cast to Domino explicitly in the Java code to be assigned to Domino variables.

Output

```
% linuxCPP
========================================
Java files compiled
Header file created
Shared library created
========================================
>>> Output <<<
d1 = <0, 0>  DOWN
d2 = <2, 6>  UP
d3 = <4, 8>  DOWN
d  = <1, 9>  UP
========================================
```

Reference Types



Arrays of Primitive Types

Arrays can be passed to native methods, can be created in native methods, and can be returned from these methods.

The JNI functions that implement arrays of primitive type use the following identifier specification and native types:

<pType>	<pjniType>	<pjniArrayType>
Boolean	jboolean	jbooleanArray
Char	jchar	jcharArray
Byte	jbyte	jbyteArray
Short	jshort	jshortArray
Int	jint	jintArray
Long	jlong	jlongArray
Float	jfloat	jfloatArray
Double	jdouble	jdoubleArray
	C types	**Java array types in C**

Main Operations

- Create a C array corresponding to a jni array, which corresponds to a Java array.

  <pjniType> * Get<pType>ArrayElements(
  JNIEnv * env, <pjniArrayType> jniArr,
  jboolean * isCopy)

  isCopy is an out parameter that indicates whether the new array is sharing the memory of the original array or is a copy of it.

- Release the C array associated with a jni array. If it is a copy, the components must be updated in the jni array.

  void Release<pType>ArrayElements(
  JNIEnv * env, <pjniArrayType> jArr,
  <pjniType> * cArr, jint mode)

  If *mode* is 0, the elements of the C array *cArr* will be copied into the jni array object *jArr*.

- Create a new jni array object.

  <pjniArrayType> New<pType>Array(
  JNIEnv * env, jsize length)

- Set elements in a jni array object from a C array buffer.

  void Set<pType>ArrayRegion(JNIEnv * env,
  <pjniArrayType> jniArr,
  jsize start, jsize length,
  <pjniType> * buffer)

- Get the length of a jarray.

  jsize GetArrayLength(JNIEnv * env, jarray ja)

Example

Pass an array of integers to a native method and have method reverse the array and return the result.

Two versions of the native code are given.

1. In the first version, the C array is reversed in place. Therefore, we need to make certain that the jni array is updated to reflect the changes made.

2. In the second version, a new C array is created and given the original components in reverse order. A new jni array object is then created and given the element of this C array.

File: ReverseA.java

```
public class ReverseA
{
    public static native int [] reverse(int [] arr);

    public static void main(String [] args)
    {
        int [] intA = new int [9];
        for (int k = 0; k < intA.length; k++)
            intA[k] = k*k;

        printA(intA);
        int [] b = reverse(intA);
        printA(b);
    }

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

    private static void printA(int [] arr)
    {
        System.out.println("Output from Java");
        for (int k = 0; k < arr.length; k++)
            System.out.print(arr[k] + " ");
        System.out.println();
    }
}
```

File: reverse.cpp (version 1)

```
#include "ReverseA.h"

JNIEXPORT jintArray JNICALL
    Java_ReverseA_reverse
        (JNIEnv * env, jclass clazz, jintArray intArr)
{
    jboolean isCopy;
    jint * ia = env->GetIntArrayElements(intArr, &isCopy);
    jsize len = env->GetArrayLength(intArr);

    for (int k = 0; k < len/2; k++)
    {
        jint temp = ia[k];
        ia[k] = ia[len-k-1];
        ia[len-k-1] = temp;
    }

    if (isCopy == JNI_TRUE)
        env->ReleaseIntArrayElements(intArr, ia, 0);

    return intArr;
}
```

File: reverse.cpp (version 2)

```
#include <stdlib.h>
#include "ReverseA.h"

JNIEXPORT jintArray JNICALL
        Java_ReverseA_reverse
                (JNIEnv * env, jclass clazz, jintArray intArr)
{
    jint * ia = env->GetIntArrayElements(intArr, NULL);

    jsize len = env->GetArrayLength(intArr);

    jint * newia = (jint*)malloc(len*sizeof(jint));

    for (int k = 0; k < len; k++)
            newia[k] = ia[len-k-1];

    jintArray newintArr = env->NewIntArray(len);

    env->SetIntArrayRegion(newintArr, 0, len, newia);

    return newintArr;
}
```

Output

```
% linuxCPP

=======================================
Java files compiled
Header file created
Shared library created
=======================================
>>> Output <<<
Output from Java
0 1 4 9 16 25 36 49 64
Output from Java
64 49 36 25 16 9 4 1 0
=======================================
```

Arrays of Objects

An array of objects can be created as a jarray in a native function.

Main Operations

- Create a jni array of objects.

```
jarray NewObjectArray(JNIEnv * env,
                      jsize length, jclass cl, jobject initVal)
```

- Set an element of an Object array.

  void SetObjectArrayElement(JNIEnv * env,
  jobjectArray joa,
  jsize index,
  jobject value)

- Get an element from an Object array.

  jobject GetObjectArrayElement(JNIEnv * env,
  jobjectArray joa,
  jsize index)

- Delete a local reference.

  void DeleteLocalRef(JNIEnv * env, jobject ob)

Example

Create an array of randomly generated dominoes in a native function and print the components of the array in Java.

File: DominoArray.java

```
public class DominoArray
{
   static
   {     System.loadLibrary("DomArray");   }

   public native static Object [] mkDominoArray(int sz);

   public static void main(String [] args)
   {
      Domino [] da = (Domino [])mkDominoArray(5);
      for (int k = 0; k < da.length; k++)
         System.out.println("elem[" + k + "] = " + da[k]);
```

```
    }
}

class Domino
{  …  }
```

File: dominoArray.cpp

```
#include "DominoArray.h"
#include <iostream.h>

JNIEXPORT jobjectArray JNICALL
                Java_DominoArray_mkDominoArray
                        (JNIEnv * env, jclass thisClass, jint sz)
{
    jclass c = env->FindClass("Domino");
    jobjectArray arr = env->NewObjectArray(sz, c, NULL);
    jmethodID cid = env->GetMethodID(c, "<init>", "(Z)V");
    cout << "Creating object array in native code.\n";
    for (int k = 0; k < sz; k++)
    {
        jobject dom = env->NewObject(c, cid, JNI_TRUE);
        env->SetObjectArrayElement(arr, k, dom);
        env->DeleteLocalRef(dom);
    }
    return arr;
}
```

Output

```
% linuxCPP

========================================
Java files compiled
Header file created
Shared library created
========================================

>>> Output <<<
Creating object array in native code.
elem[0] = <0, 8>  UP
elem[1] = <7, 8>  UP
elem[2] = <4, 9>  UP
elem[3] = <1, 5>  UP
elem[4] = <2, 7>  UP

========================================
```

Type jvalue

When calling a Java method or constructor from a native function, parameters can be passed as a C array of jvalue.

The type jvalue is defined as follows:

```
typedef union jvalue
{
    jboolean    z;
    jchar       c;
    jbyte       b;
    jshort      s;
    jint        i;
    jlong       j;
    jfloat      f;
    jdouble     d;
    jobject     o;
} jvalue;
```