JNI: JAVA와 C++의 연동

출처: C++ for Java Programmers
저자: Mark Allen Weiss
번역 및 추가: Frank Yoon (moses@maru.net)
테스트 환경: Windows XP, Java 1.5, Visual Studio 6.0, Vim 7.x

JVM에서 네이티브 함수는 C나 C++과 같은 언어로 구현됩니다. JDK는 Java Native Interface라고 불리는 표준 프로그래밍 인터페이스를 지원합니다. 즉, 이론적으로 JVM에서 C나 C++ 코드를 다소 이식성 있게 호출할 수 있습니다.

■ JNI 기본

네이티브 메소드(native method)란 자바에서 특화된 메소드로써, 다른 언어로 구현되어 있으며 native라는 예약어를 가지는 메소드를 말합니다.

다음은 네이티브 메소드를 이용하는 몇 가지 공통적인 이유입니다.

1. 우리가 이미 다른 언어로 상당히 크고 중요한 코드를 작성했고, 자바로 동일한 코드를 재작성하기를 원치 않으며, 단지 기존 코드를 재사용하고 싶을 때 JNI를 사용합니다.
2. 우리가 시스템 디바이스에 접근할 필요가 있거나 혹은 자바의 성능을 넘어선 플랫폼 특정적인 작업을 수행할 경우에 JNI를 사용할 수 있습니다. 사실 많은 자바 라이브러리 루틴들이 결과적으로는 내부(private) 네이티브 메소드를 이러한 목적으로 호출하고 있습니다. 예를 들면, I/O, threading, networking 패키지들 전부가 내부 네이티브 메소드를 포함합니다.
3. 자바로 구현하기엔 어플리케이션이 매우 느릴 수 있으며, C++에서 time-critical 코드를 구현함으로써 성능향상을 가져올 수 있을 경우에 JNI를 사용합니다.

그러나 JNI에는 다음과 같은 다소 부정적인 면도 있습니다.

1. 이식성(Portability)을 잃게 됩니다. 많은 자바 라이브러리가 네이티브 메소드를 사용하고 있는데, 새로 추가된 네이티브 메소드가 모든 플랫폼에서 호출 가능한 동일한 C++ 코드를 가진다면, 그 코드는 자바에서 가장 먼저(in the first place) 구현되어질 수 있습니다.
2. 안정성(safety)를 잃게 됩니다. 네이티브 메소드는 자바 메소드처럼 동일한 보호를 보장받지 못합니다. 일단 C++ 코드에 들어가게 되면, 자바에서의 모든 것이 백지화 되고 메모리 추적, 포인터의 사용 그리고 배열 바운딩에 의한 C++ 버그들이 발생할 수 있습니다.
3. 네이티브 메소드의 구현은 .dll 및 .so와 같은 동적 라이브러리들 내에서 이루어집니다. 네이티브 메소드를 사용하는 자바 코드는 반드시 동적 라이브러리들을 호출하는데, 이는 자바 시큐리티 매니저에 반대되는 연산입니다. 일례로 애플릿에서는 일반적으로 사용자 정의 네이티브 메소드를 호출할 수 없습니다.
4. 코드 자체가 방해물이 될 수 있습니다. 타이핑 오류와 같은 컴파일 오류에 종종 직면하게 됩니다.

JNI를 사용하는 기본절차는 다음과 같습니다.

1. native를 표시함으로써 자바로 구현되지 않는 특정 메소드들을 가지는 자바 클래스를 선언한다.
2. JNI 프로토콜을 사용하여 네이티브 메소드를 구현하는 C++ 함수를 작성합니다.

3. C++ 함수를 컴파일하고 동적 라이브러리에 삽입합니다.
4. JVM이 동적 라이브러리를 호출합니다. 네이티브 메소드에 대한 호출은 동적 라이브러리 내에서 구현된 호출에 의해 다루어집니다.

JNI는 C++뿐만 아니라 객체 지향 언어가 아닌 C나 다른 언어를 통해서도 구현될 수 있습니다. 따라서 다음과 같은 의문들이 생길 수 있습니다.

1. 자바 객체에서 사용되는 필드나 메소드는 클래스가 없는 C에서는 어떻게 되는가?
2. 매개변수는 자바로부터 C/C++로 어떻게 전달하는가?
3. 리턴값은 C/C++로부터 자바로 어떻게 전달되는가?
4. C에서 허용되지 않는 함수 오버로딩은 어떤가?
5. 정적 멤버와 일반 멤버를 어떻게 구별하는가?
6. 문자열과 배열은 어떻게 되는가?
7. 어떤 방법으로 C/C++ 코드가 예외(exception)를 던질 수 있는가? 그리고 예외를 발생하는 자바 메소드를 호출하게 되면 무슨 일이 발생하는가?

■ 매개변수가 없고 반환값이 없는 간단한 메소드의 구현

다음과 같이 간단한 네이티브 메소드를 선언해봅시다.

```
class HelloNative
{
	native public static void hello();

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

class HelloNativeTest
{
	public static void main(String[] args)
	{
		HelloNative.hello();
	}
}
```

먼저, HelloNative 클래스는 단일 네이티브 메소드 hello()를 지니고 있습니다. 그리고 테스트 프로그램 HelloNativeTest에서 이 네이티브 메소드를 호출하고 있습니다.

호출할 메소드가 네이티브 이므로, JVM은 로드된 동적 라이브러리들 중에서 호출할 메소드를 찾습니다.

System.loadLibrary()를 사용하여 네이티브 메소드를 구현한 동적 라이브러리를 JVM으로 로드합니다. 호출할 때 해당 라이브러리가 로드되지 않거나, 로드할 동적 라이브러리의 이름이 잘못되거나 혹은 라이브러리가 자바 속성값 java.library.path 에 설정된 디렉토리에 위치하지 않을 경우엔UnsatisfiedLinkError가 발생합니다.

다음으로, 동적 라이브러리를 생성하기 위해 C++ 코드를 작성해야 합니다. 자바에서 네이티브 메소드 이름이 hello()라고 해서 C++에서도 hello()라는 함수를 구현해야 하는 것은 아닙니다. JNI는 완전한 클래스 이름, 메소드 이름, 그리고 자바 네이티브 메소드의 매개변수 타입에 기초한 함수를 제공하는 복잡한 알고리즘을 명세하고 있습니다. 그러나 우리는 이 알고리즘에 대해서 알 필요가 없습니다. 대신, 우리는 javah 유틸리티를 이용하여 자바 클래스로부터 C++ 헤더 파일을 생성할 수 있습니다.

위의 예제에 대해선 다음을 실행합니다.

```
javah HelloNative
```

생성된 헤더파일(HelloNative.h)은 다음과 같습니다.

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


#ifndef _Included_HelloNative
#define _Included_HelloNative
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     HelloNative
 * Method:    hello
 * Signature: ()V
 */
JNIEXPORT void JNICALL Java_HelloNative_hello
  (JNIEnv *, jclass);


#ifdef __cplusplus
}
#endif
#endif
```

먼저 위 예제에서 **Java_HelloNative_hello** 함수를 볼 수 있습니다. 이는 자바에서의 디폴트 패키지, 클래스 그리고 메소드 이름을 반영한 것입니다.

그리고 이 함수는 두 개의 매개변수를 가집니다. 첫 번째 매개변수는 JNIObject 에 대한 포인터로써 JNIObject 객

체들의 필드와 메소드를 접근하는 데 광범위하게 사용됩니다. 두 번째 매개변수는 jclass 객체로써 HelloNative 클래스에 대한 정보를 나타냅니다. Reflection API에 익숙한 자바 프로그래머라면 Class 오브젝트와 동일한 것으로 간주할 수 있습니다.

마지막으로 네이티브 메소드가 특정 매개변수를 취하지 않으므로, C++ 선언에서도 처음의 두 매개변수를 제외하고 추가적인 매개변수는 나열되지 않습니다.

위 헤더파일에 대한 구현(HelloNative.cpp)은 다음과 같으며, 아래와 같이 javah에 의해 생성된 헤더파일을 추가되어 있습니다.

```
#include <iostream>
using namespace std;
#include "HelloNative.h"
JNIEXPORT void JNICALL Java_HelloNative_hello(JNIEnv *env, jclass cls)
{
        cout << "Hello world"<< endl;
}
```

■ 공유 라이브러리 컴파일

이제 공유 라이브러리 속으로 컴파일하는 것만 남았습니다. 이는 시스템환경에 따라 좌우됩니다. 헤더파일이 jni.h 파일을 include 하고 있으므로, 컴파일시 이를 반드시 포함해야 합니다. (%JAVA_HOME%₩include₩jni.h)

윈도우 환경에서는 컴파일은 다음과 같습니다. (Visual Studio의 커맨드라인 컴파일러를 사용)

```
cl -GX /GR -I%JAVA_HOME%₩include -I%JAVA_HOME%₩include₩win32 HelloNative.cpp -LD -FeHelloNative.dll
```

유닉스 환경에서는 다음과 같습니다. 솔라리스의 경우, linux를 solaris로 대체하면 됩니다.

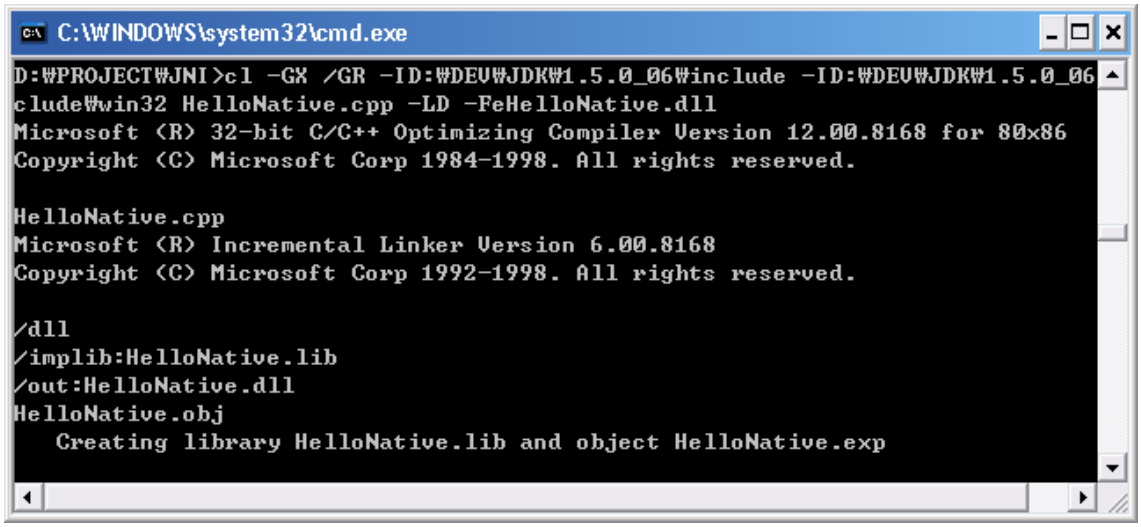
```
g++ -c –fPIC –I$JAVA_HOME/include –I$JAVA_HOME/include/linux HelloNative.cpp
g++ -shared –o libHelloNative.so HelloNative.o
```

Standard Sun C++ 컴파일러를 사용할 경우, 다음과 같습니다.

```
CC –G –I$JAVA_HOME/include –I$JAVA_HOME/include/solaris HelloNative.cpp –o libHelloNative.so
```

유닉스에서는 동적 라이브러리는 JVM이 호출될 때의 현재 디렉토리나 혹은 LD_LIBRARY_PATH 환경변수에 리스트된 디렉토리에 위치해야 합니다. 윈도우의 경우에도 동적 라이브러리는 현재 디렉토리나 혹은 PATH 환경변수에 리스트된 디렉토리에 위치해야 합니다.

다음은 컴파일 및 실행결과입니다.



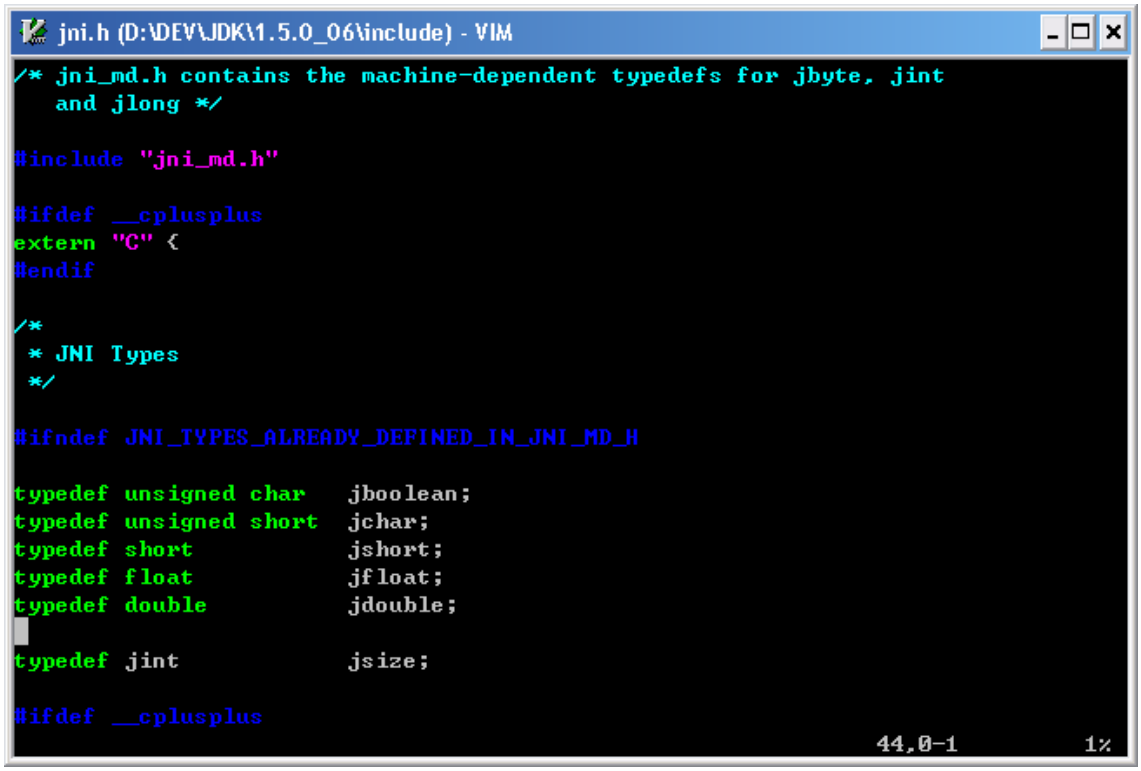
■ JNI Types

jni.h 헤더 파일은 다음과 같이 타입 군을 정의하고 있는데, 모두 j 문자로 시작합니다. 기본적인 8개의 자바 primitive 타입과 대응하는 c++ 타입은 다음과 같습니다. 이러한 것들은 **jbyte, jshort, jint, jlong, jfloat, jdouble, jchar** 및 **jboolean** 등이 있습니다. 기계 의존적인 타입인 jbyte, jint, jlong은 윈도우일 경우 %JAVA_HOME%/include/win32/jni_md.h 등에 선언되어 있습니다. 64비트 머쉰에서는 jlong 은 곧잘 int로써 typedef 되어질 수 있습니다. 또 jni.h 헤더 파일에는 C++에서의 size_t와 같은 **jsize** 타입도 정의하고 있습니다.

자바에서 String과 Object와 대응하는 C++타입으로써 **jstring**과 **jobject** 타입이 있습니다. C++ JNI 루틴 호출에 의해 const char* 로부터 jstring 객체가 생성할 수 있으며, 비슷하게 jstring 객체로부터 char* 를 생성할 수 있습니다. 자바의 Array 객체도 대응되는 C++타입에 의해 표현할 수 있습니다. 이러한 것들에는 **jintArray**와 같이 primitive 형태의 배열뿐만 아니라 **jobjectArray**와 같은 클래스 형태의 배열 등이 있습니다.

마지막으로 메소드와 필드에 대해 정보를 표현하는 데 사용하는 두 객체들이 있습니다. 이러한 것들로는 **jmethodId**와 **jfieldID**와 같은 것들이 있으며, 자바 Reflection API에서의 Method 및 Field와 개념적으로 같은 것들입니다.

이상, jni.h 파일의 내용은 다음과 같습니다.



■ 자바 객체에 대한 접근

네이티브 코드에 의해 자바 객체가 어떻게 다루어지는 지 알아보기 위해, 다음과 같이 printDate 메소드를 제외한 완벽한 Date 클래스를 구현해보기로 합니다. 분명한 것은 printDate 인스턴스 메소드의 구현은 인스턴스 데이터에 억세스 하거나 혹은 Date 인스턴스의 어느 메소드들을 호출해야 한다는 것입니다.

먼저 데이터 멥버에 엑세스하는 코드를 작성하고, 다음으로는 getMonth, getDay 그리고 getYear와 같은 접근자를 호출하는 코드를 작성해보겠습니다. 마지막으로 toString 메소드를 호출하여 jstring으로부터 어떻게 C-style 문자열을 추출할 수 있는지 설명하겠습니다.

네이티브 메소드 printDate를 가지는 Date 클래스는 다음과 같습니다.

```
class Date
{
        public Date(int m, int d, int y)
        {month=m; day=d; year=y;}

        Static
        {
                System.loadLibrary(“Date”);
        }

        native public void printDate();

        public int getMonth()
```

```
        { return month;}
        public int getDay()
        { return day;}
        public int getYear()
        { return year;}

        public String toString()
        { return month + "/" + day + "/" + year; }

        private int month;
        private int day;
        private int year;
}
```

위 Date 클래스에 대한 테스트 클래스는 다음과 같습니다.

```
class TestDate
{
        public static void main(String[] args)
        {
                Date d = new Date(3, 1, 2006);
                d.printDate();
        }
}
```

A. 필드에 대한 접근

이상 기술된 Date 클래스에 대한 javah의 실행 결과로 얻어지는 헤더 파일은 다음과 같습니다.

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class Date */
#ifndef _Included_Date
#define _Included_Date
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     Date
 * Method:    printDate
 * Signature: ()V
 */
```

```
JNIEXPORT void JNICALL Java_Date_printDate (JNIEnv *, jobject);
#ifdef __cplusplus
}
#endif
#endif
```

printDate 메소드에 대해 선언된 함수 Java_Date_printDate 의 매개변수 jobject에 대해 주목할 필요가 있습니다. 이는 정적 메소드가 아닌 인스턴스 메소드를 구현할 것임을 나타내고 있습니다.

jobject의 필드에 억세스하기 위해서는 다음의 단계를 거쳐야 합니다.

1. 객체에 대한 클래스 정보를 표현하는 jclass를 획득한다.
2. jclass 내의 특정 필드에 대한 필드 정보를 표현하는 jfieldID를 획득, 이를 위하여 필드의 이름과 타입을 알아야 한다.
3. jfieldID와 특정 객체에 대한 필드를 획득하기 위한 jobject를 전달하여 메소드를 호출한다.

가장 꽁수는 jfieldID를 획득하는 과정입니다. 필드의 이름은 쉬운 편이나, 필드의 타입은 다른 클래스 타입의 배열과 같은 모호한 타입이 될 수 있으므로 까다롭습니다. 따라서 자바는 모든 필드의 타입과 메소드 시그너쳐를 생성하기 위한 유틸리티를 제공하고 있습니다. 이를 위해 해당 클래스는 사전에 컴파일 되어야 합니다. javap 유틸리티는 완전한 리스팅을 위해 두 가지 옵션을 취합니다.

본 Date 예제를 위한 커맨드 명령은 다음과 같습니다.

```
javap –s –private Date
```

위의 결과물은 다음과 같습니다.

```
#>javap -s -private Date
Compiled from "Date.java"
class Date extends java.lang.Object{
private int month; Signature: I
private int day; Signature: I
private int year; Signature: I
public Date(int, int, int); Signature: (III)V
public native void printDate(); Signature: ()V
public int getMonth(); Signature: ()I
public int getDay(); Signature: ()I
public int getYear(); Signature: ()I
public java.lang.String toString(); Signature: ()Ljava/lang/String;
static {}; Signature: ()V
}
```

위에서 보다시피 반환값이 int 타입일 경우 필드타입은 I, 반환값이 void 타입일 경우 필드타입은 V, 반환값이

java.lang.String 타입일 경우 필드타입은 Ljava/lang/String으로 나타냅니다. 따라서 getYear 메소드에 대해 ()I, printDate 메소드에 대해 ()V, 그리고 toString 메소드에 대해 ()Ljava/lang/String를 각각 필드 타입으로 가지게 됩니다. String 배열일 경우에는 [Ljava/lang/String;으로 인코딩됩니다.

이상 위 시그너처 사전준비를 이용하여 printDate 코드의 구현은 다음과 같습니다.

```
//printDate()의 첫 번째 구현
#include "Date.h"
#include <iostream>
using namespace std;

JNIEXPORT void JNICALL Java_Date_printDate(JNIEnv *env, jobject ths)
{
	jclass cls = env->GetObjectClass(ths);

	jfieldID monthID = env->GetFieldID(cls, "month", "I");
	jfieldID dayID   = env->GetFieldID(cls, "day", "I");
	jfieldID yearID  = env->GetFieldID(cls, "year", "I");

	jint m = env->GetIntField(ths, monthID);
	jint d = env->GetIntField(ths, dayID);
	jint y = env->GetIntField(ths, yearID);

	cout << m << "/" << d << "/" << y << endl;
}
```

jclass는 ths 엔티티에 의해 획득할 수 있고, 이는 Date 클래스에 대한 정보를 가집니다. 코드 전체를 통해 JNIEnv 내의 멤버 함수 호출을 볼 수 있습니다. 우리는 이를 환경 멤버 함수(environment member functions)라고 부릅니다. JNI에 의해 제공되는 거의 모든 함수들이 같은 방식으로 env 포인터를 통해 호출됩니다.

jclass를 가지게 되면, 클래스와 필드명, 필드 타입을 전달함으로써 GetFieldID 호출에 의해 세 개의 데이터 필드에 대한 jfieldID들을 획득할 수 있습니다. 마지막으로 객체와 jfieldID를 지정함으로써 객체의 필드에 접근할 수 있습니다.

8개의 primitive 타입에 대해 각각 GetXXXField라는 메소드 군이 존재합니다. 마찬가지로 Object에 대한 GetObjectField 함수도 있습니다. String 객체는 GetObjectField에 의해 접근하여 jstring으로 다운캐스팅됩니다. 심지어 SetXXXField 메소드를 호출하여 Field 값을 변경할 수 있습니다.

정적 필드는 인스턴스 필드와 다르게, GetFieldID 대신에 GetStaticFieldID 를 사용하고, jobject 엔티티 ths 대신에 jclass 엔티티 cls 를 매개변수로 전달하여 GetStaticXXXField를 호출합니다. 마찬가지로 SetStaticXXXField를 이용하여 정적 필드의 값을 변경합니다.

String 필드는 GetObjectField 메소드를 이용하여 접근하고, 결과 jobject 는 반드시 jstring 으로 형변환되어야 합니다. dynamic_cast 연산자를 이용하여 동적 변환이 이루어질 수 있으나 때때로 jobject 가상 메소드를 전혀 선언하지 않았을 경우엔 reinterpret_cast 연산자를 이용하여 강제적으로 형변환을 시도할 수 있습니다.

B. 메소드 호출

메소드 호출 역시 필드 접근과 비슷한 방법으로 이루어집니다. 먼저, 호출할 메소드가 속해 있는 해당 jclass 엔티티가 필요합니다. 다음으로, 메소드를 표현하는 jmethodID를 획득합니다. 이를 위해 jclass 엔티티 그리고 메소드 이름 그리고 시그너처가 필요합니다. 마지막으로 메소드를 호출합니다. GetXXXField 메소드 군과 마찬가지로 메소드를 호출하기 위한 다음의 메소드 군이 있습니다. 여기서, ...는 메소드에 대한 매개변수들이며, XXX는 리턴 타입입니다.

```
XXX CallXXXMethod(jobject ths, jmethodID m, ...);
XXX CallStaticXXXMethod(jclass cls, jmethodID m, ...);
```

다음은 printDate()의 두번째 구현입니다.

```
//printDate()의 두 번째 구현
#include "Date.h"
#include <iostream>
using namespace std;

JNIEXPORT void JNICALL Java_Date_printDate(JNIEnv *env, jobject ths)
{
        jclass cls = env->GetObjectClass(ths);
        jmethodID getMonthID, getDayID, getYearID;

        getMonthID = env->GetMethodID(cls, "getMonth", "()I");
        getDayID   = env->GetMethodID(cls, "getDay", "()I");
        getYearID  = env->GetMethodID(cls, "getYear", "()I");

        jint m = env->CallIntMethod(ths, getMonthID);
        jint d = env->CallIntMethod(ths, getDayID);
        jint y = env->CallIntMethod(ths, getYearID);

        cout << m << "/" << d << "/" << y << endl;
}
```

CallXXXMethod 의 호출은 동적 배정(dynamic dispatch)를 사용할 수 있습니다. 다음 메소드는

```
XXX CallNonvirtualXXXMethod(jobject ths, jmethodID m, ...);
```

동적 배정을 사용할 수 없으나 사용할 일은 거의 없습니다.

C. 생성자 호출

생성자는 NewObject 환경변수 메소드를 사용하여 호출할 수 있습니다. 이를 위하여 jclass, jmethodID 및 매개변수를 전달합니다. jmethodID는 "<init>" 를 메소드 이름으로 사용합니다. 네이티브 단에서 Date 객체를 생성하는 법은 다음과 같습니다.

```
jclass dateClass = env->FindClass("Date");
jmethodID ctor = env->GetMethodId(dateClass, "<init>", "(III)V");
jobject d1 = env->NewObject(dateClass, ctor, 1, 1, 2006);
```

■ 문자열 그리고 배열

jstring 및 jintArray와 같은 다양한 배열 오브젝트는 C++ 코드에서 바로 사용될 수 없습니다. 따라서 environment 함수가 적절한 JNI 타입으로부터 primitive 문자열이나 배열을 생성하도록 해야 합니다.

A. 문자열

jstring 형으로부터 const char * 형 문자를 획득하기 위해서 environment 함수 GetStringUTFChar를 호출해야 합니다. GetStringUTFChar 매개변수는 jstring과 jboolean 값의 주소입니다. 만약 주소가 NULL이 아니면, Boolean 값은 char *가 원본의 복사본일 경우 true로 설정되고, 어떤 방법으로든지 복사본이 만들어지지 않으면 false로 설정됩니다. 사실 복사본의 생성 여부는 중요하지 않으므로, 일반적으로 NULL을 두 번째 매개변수로 전달합니다.

메모리 힙으로부터 GetStringUTFChars 메소드에 의해 할당된 const char * 작업이 끝나면, 반드시 ReleaseStringUTFChars 메소드로 해제를 해주어야 메모리 누출을 방지할 수 있습니다.

다음은 printDate()의 세번째 구현입니다. JVM으로부터 toString()을 호출하였습니다.

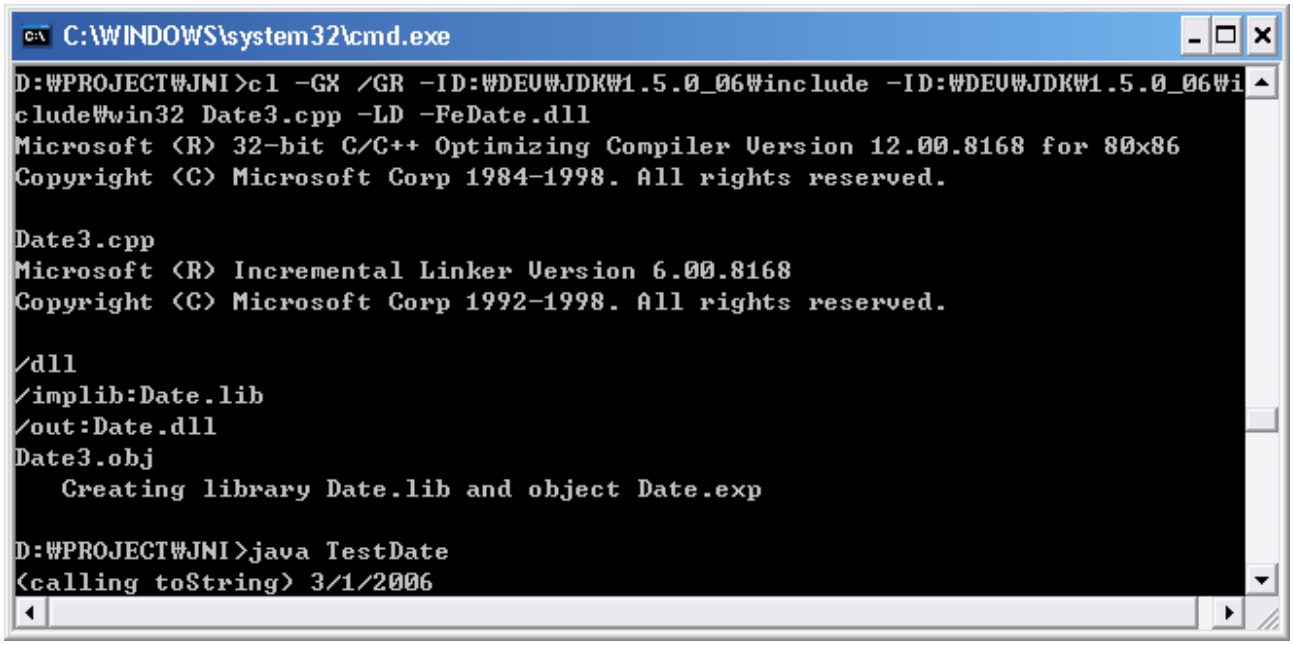
```
//printDate()의 세 번째 구현
#include "Date.h"
#include <iostream>
using namespace std;

JNIEXPORT void JNICALL Java_Date_printDate(JNIEnv *env, jobject ths)
{
        jclass cls = env->GetObjectClass(ths);

        jmethodID toStringID = env->GetMethodID(cls, "toString", "()Ljava/lang/String;");
        jstring str = (jstring) env->CallObjectMethod(ths, toStringID);

        const char *c_ret = env->GetStringUTFChars(str, NULL);
        cout << "(calling toString) " << c_ret << endl;
        env->ReleaseStringUTFChars(str, c_ret);
}
```

다음은 실행결과입니다.



또한 NewStringUTF 를 호출하여 새로운 jstring 객체를 추가할 수 있습니다. 다음은 StringAdd 클래스의 문자열을 연결하는 정적 메소드 add를 네이티브 코드로 구현한 예제입니다.

```
//문자열 연결
JNIEXPORT jstring JNICALL Java_StringAdd_add(JNIEnv *env, jclass cls, jstring a, jstring b)
{
        const char *a1 = env->GetStringUTFChars(a, NULL);
        const char *b1 = env->GetStringUTFChars(b, NULL);
        char *c = new char[strlen(a1) + strlen(b1) + 1];

        strcpy(c, a1);
        strcpy(c, b1);
        jstring result = env->NewStringUTF(c);

        env->ReleaseStringUTFChars(a, a1);
        env->ReleaseStringUTFChars(b, b1);
        delete [] c;

        return result;
}
```

B. 배열

전형적인 배열은 jintArray이고, Object 배열을 표현하는 jobjectArray 배열이 추가적으로 있습니다. environment 함수 GetArrayLength는 배열 객체의 길이를 반환합니다. 배열의 개별 아이템을 접근하는 것은 동일합니다.

다음은 배열의 합을 구하는 자바 클래스입니다. sum 메소드가 네이티브로 선언되어 있습니다.

```
class NativeSumDemo
{
	native public static double sum(double[] arr);

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

	public static void main(String[] args)
	{
		double [] arr = { 3.0, 6.5, 7.5, 9.5};
		System.out.println(sum(arr));
	}
}
```

다음은 javah 에 의해 생성된 헤더파일입니다. (NativeSumDemo.h)

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class NativeSumDemo */
#ifndef _Included_NativeSumDemo
#define _Included_NativeSumDemo
#ifdef __cplusplus
extern "C" {
#endif

/*
 * Class:     NativeSumDemo
 * Method:    sum
 * Signature: ([D)D
 */
JNIEXPORT jdouble JNICALL Java_NativeSumDemo_sum  (JNIEnv *, jclass, jdoubleArray);
#ifdef __cplusplus
}
#endif
#endif
```

sum 메소드가 double형을 반환값으로 가지는 static 메소드이므로, 네이티브단에서 Java_NativeSumDemo_sum 함

수의 매개변수로 jclass, jdoubleArray 가 옵니다.

다음은 네이티브 메소드 Java_NativeSumDemo_sum의 구현입니다.

```
#include "NativeSumDemo.h"
JNIEXPORT jdouble JNICALL Java_NativeSumDemo_sum(JNIEnv *env, jclass cls, jdoubleArray arr)
{
        jdouble sum = 0;
        jsize len = env->GetArrayLength(arr);

        //Get the elements; don't care to know if copied or not
        jdouble *a = env->GetDoubleArrayElements(arr, NULL);

        for(jsize i = 0; i < len; i++)
                sum += a[i];

        // Release elements; no need to flush back
        env->ReleaseDoubleArrayElements(arr, a, JNI_ABORT);

        return sum;
}
```

sum 은 jdouble 로 선언되어 있고 0으로 초기화 되었습니다. 배열의 길이는 GetArrayLength 메소드를 통해 얻을 수 있습니다. 그리고 배열 jdoubleArray 로부터 C 스타일의 배열을 획득합니다. 문자열의 경우, 이 배열은 VM의 구현에 따라 복사가 되거나 혹은 원본 그 자체일 수 있습니다. 배열에 대해선, double 과 jdouble 이 개념적으로 동일하고, 자바 배열이 순차적으로 저장될 경우엔, 포인터 변수가 VM내의 원본 double[] 을 저장하고 있는 실제 메모리 위치를 가리킬 수 있고, 이 경우엔 배열 복사는 이루어지지 않습니다.

그러나 배열에 대한 포인터 변수가 한번 제출이 되면, 가비지 콜렉터는 이 포인터를 무효화하지 않고선 배열의 일부분을 안전하게 이동할 수 없게 됩니다. 따라서 복사가 이루어지지 않을 경우, 원본 배열은 고정이 되어 배열에 대한 포인터가 해제될 때까지 재배치할 수 없게 됩니다.

배열이 복사본일 경우, 첫째로 우리는 어떠한 변경일지라도 원본으로 복사해 넣어야 합니다. 그렇지 않을 경우, 배열에 대한 변경은 반영되지 않습니다. 둘째로, 반드시 메모리를 회수해야 합니다. 그 결과로 ReleaseXXXArrayElements 의 마지막 매개변수가 0, JNI_COMMIT, 혹은 JNI_ABORT가 될 수 있습니다. 매개변수가 0일 경우, 원본으로 모든 컨텐츠를 플러쉬하게 되므로 배열에 대한 모든 변경이 반영되고, 필요하다면 메모리를 회수합니다. JNI_COMMIT는 컨텐츠를 플러쉬하나 메모리는 회수하지 않습니다. 배열에 대한 변경이 즉시로 반영되어야 할 필요가 있으나 복사본이 만들어지지 않았을 경우, 유용한 옵션입니다. JNI_ABORT는 컨텐츠를 플러쉬하지는 않고, 단지 필요하다면 메모리만 회수합니다. 이는 배열에 대한 수정 작업이 하지 않을 경우에 유용합니다.

jobjectArray 내의 요소들에 대한 접근은 보다 복잡합니다. 왜냐하면 C스타일과 같은 형태의 요소들을 획득할 수

없기 때문입니다. 대신 env 포인터를 통해 다음을 호출하여 접근할 수 있습니다.

GetObjectArrayElement(array, idx);

SetObjectArrayElement(array, idx, val);

새 배열의 생성은 다음의 메소드를 사용합니다. 반환값은 jarray 타입입니다.

NewObjectArray(jclass cls, int len, jobject default);

NewXXXArray(int len, XXX default);

■ 예외처리

네이티브 메소드는 예외를 던질필요가 있을 경우, environment 메소드의 Throw 나 ThrowNew를 호출합니다. ThrowNew가 용법이 더욱 쉽습니다. 예외클래스의 이름과 매개변수를 생성자에게 넘겨주면 됩니다.

```
#include <iostream>
using namespace std;

#include "NativeSumDemo.h"
JNIEXPORT jdouble JNICALL Java_NativeSumDemo_sum(JNIEnv *env, jclass cls, jdoubleArray arr)
{
	jdouble sum = 0;
	jsize len = env->GetArrayLength(arr);

	if(len == 0)
	{
		env->ThrowNew(env->FindClass("java/lang/Exception"), "Empty array");
		cout << "Throwing an exception, but should exit" << endl;
		return 0.0;
	}

	// Get the elements; don't care to know if copied or not
	jdouble *a = env->GetDoubleArrayElements(arr, NULL);
	for(jsize i = 0; i < len; i++)
		sum += a[i];

	// Release elements; no need to flush back
	env->ReleaseDoubleArrayElements(arr, a, JNI_ABORT);
	return sum;
}
```

■ JAVA 모니터

만약 네이티브 메소드가 synchronized 로 선언되면, 일반적인 모니터 객체[인스턴스 메소드에 대해 this, 정적 메

소드에 대해 getClass()]를 메소드가 리턴되기 전에 획득할 수 있습니다. 만약 메소드가 synchronized 되지 않거나, synchronized 되기에 불충분할 경우에도 모니터 객체를 획득할 수 있습니다. Environment 객체의 MonitorEnter나 MonitorExit 메소드를 통해 네이트비 코드에서 동기화 블록을 구분지을 수 있습니다.

따라서 다음은

```
env->MonitorEnter(obj);
/* synchronized block */
env->MonitorExit(obj);
```

아래와 동일합니다.

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

wait 나 notifyAll 과 같은 메소드는 jmethodID 등을 획득하는 일반적인 JNI 메커니즘을 통해 호출할 수 있습니다.

■ Invocation API

Invocation API는 C++ 프로그램 내에서 가상 머쉰(Virtual Machine)을 생성할 수 있게 합니다. VM이 한번 생성되면 클래스에 관계없이 일반적인 메커니즘이 main 메소드를 호출하는 데에 사용될 수 있습니다.

다음은 그 예제 코드입니다.

```
//Invoke the main method of Hello
#include <iostream>
#include <jni.h>
using namespace std;

const char *CLASS_NAME = "Hello";

int main(int argc, char *argv[])
{
        JavaVMInitArgs vm_args;
        JavaVMOption options[1];
        JavaVM *vm;
        JNIEnv *env;

        options[0].optionString = "-Djava.class.path=.";
        vm_args.options = options;
        vm_args.nOptions = 1;
```

```
	vm_args.version = JNI_VERSION_1_2;

	int res = JNI_CreateJavaVM(&vm, (void**)&env, &vm_args);
	if(res < 0)
	{
		cerr << "Failed to create VM (" << res << ")" << endl;
		return -1;
	}

	jclass cls = env->FindClass(CLASS_NAME);

	// Now try to call main
	jmethodID mainMethodID = env->GetStaticMethodID(cls, "main", "([Ljava/lang/String;)V");
	jclass classString = env->FindClass("java.lang.String");
	jobjectArray argsToMain = env->NewObjectArray(0, classString, NULL);
	env->CallStaticVoidMethod(cls, mainMethodID, argsToMain);

	vm->DestroyJavaVM();

	return 0;
}
```

코드 전체는 main 메소드를 호출하기 위한 커맨드 매개변수들을 초기화하고, 또 실제 호출하는 과정들입니다. 이 코드를 컴파일 하기 위해서는 jvm.lib (Window) 혹은 jvm (Unix) 라이브러리가 링크되어 있어야 합니다.