제 11 장 고유 메소드

- 자바에서 C 함수 호출하기
- 숫자 전달인자와 반환값
- 문자열 전달인자
- 객체 필드 접근하기
- 정적 필드 접근하기
- 서명
- 자바 메소드 호출하기
- 배열
- 오류 처리
- 호출 API
- 완전한 자바 예제: 윈도우 레지스트리에 접근하기

우리는 여러분이 자바 프로그래밍 언어로 작성된 코드가 C 또는 C++와 같은 플랫폼에 종속적인 언어로 쓰여진 코드에 보다 많은 장점들을 갖는 다는 사실을 확신하리라 희망한다. 물론 다음은 이식성에 대한 이슈라기 보다는 기능들에 대한 것이다.

- 여러분은 C 나 C++보다 자바 프로그래밍 언어로 버그 없는 코드를 작성하기 쉽다.
- 멀티쓰레딩은 아마도 다른 언어들보다 자바 프로그래밍 언어로 코드하기가 더 쉽다.
- 네트워킹 코드가 쉽다.

이식성은 단지 코드 라인에서 얻을 수 있는 보너스와 같은 것이다.

실제적으로 "100% 순수 자바" 솔루션은 응용 프로그램에 대해서 아주 좋은 원칙이다. 여러분은 다른 언어로 작성된 코드(일반적으로 네이티브 코드라 불림)를 작성(또는 사용)할 필요가 있는 상황이 발생한다. 이렇게 하길 원하는 데에는 명백히 세가지 이유가 있다.

1. 다른 언어로 작성되었으며 충분히 검증이 된 상당히 많은 양의 코드가 존재한다. 이 코드를 자바로 이식하는 것은 시간이 많이 걸리며, 자바로 이식된 코드는 다시 디버깅을 해야 한다.
2. 응용 프로그램이 시스템 기능이나 디바이스에 접근할 필요가 있으며 자바를 사용할 경우 번거롭거나 불가능하다.
3. 코드의 속도를 최대로 하는 것이 매우 중요하다. 작업에 걸리는 시간이 결정적으로 중요하거나, 아주 자주 사용되는 코드이므로 최적화하였을 경우 큰 이점을 가진다. (물론, JIT (just in time) 컴파일을 사용하면 이런 경우가 덜 발생한다. JIT 로 처리한 후에는 자바의 집중적인 계산 수행도 컴파일된 C 코드에 비해 그다지 느리지 않다.)

만일 여러분이 위의 세가지 상황중 하나를 만난다면, 자바 프로그래밍 언어로 작성된 프로그램에서 네이티브 코드를 호출하는 것은 당연한 것이다. 물론, 자바의 보안 제한 때문에 일단 고유 코드를 사용하기 시작하면 애플릿 대신에 자바 응용 프로그램으로 제한을 받게 된다. 특히, 호출되는 라이브러리는 클라이언트 기계에 놓여있어야 할 것이다.

고유 메소드를 호출하게 하는 것은 가능하다, 자바 기술은 시스템 라이브러리들과 동작하기 위한 상황도 필요하고 JDK 는 단조로운 프로그래밍을 덜어주는 몇가지 도구를 가지고 있다.

노트 : 여러분의 고유 코드를 위해 사용하는 언어는 C 나 C++이 아니어도 된다. 만일 여러분이 자바와 포트란 프로그래밍 언어사이의 바인딩에 접근한다면 여러분은 포트란 컴파일러로 컴파일된 코드를 사용할 수 있다.

다음을 명심하라: 고유 메소드를 사용하면 이식성을 잃게 된다. 이것은 여러분의 프로그램을 응용 프로그램 형태로 배포할 때에도 마찬가지이다. 여러분이 지원하고자 하는 각 플랫폼에 대해 독자적인 고유 메소드 라이브러리를 제공하여야만 한다. 이것은 사용자에게 이 라이브러리를 설치하는 방법도 알려주어야만 한다는 것을 의미한다. 또한, 사용자가 애플릿이 데이터에 해를 끼치지 않으며 비밀 자료를 누출하지 않는다는 사실을 믿는다 하더라도, 그 사용자는 고유 메소드 라이브러리를 사용한 자바 코드는 믿지 않을 수 있다. 이런 이유로 인하여, 많은 잠재적인 사용자들이 고유 코드를 필요로 하는 자바 프로그램 사용을 주저할 것이다. 이러한 보안 문제와는 별도로, 고유 라이브러리는 자바 코드보다 안전하지 못하다. 특히, C 또는 C++로 작성된 고유 라이브러리는 부적절한 포인터를 통한 메모리 덮어쓰기에 대한 보호를 하지 못한다. 자바 가상 기계를 망가지게 하거나 그 보안을 손상하거나 운영 체제를 다운 시키는 고유 메소드 작성이 쉽다.

따라서, 고유 코드의 사용은 최후의 수단이 되도록 하는 것이 좋다. 만일 자바 프로그램에서 프린터나 직렬 포트와 같은 디바이스에 접근을 원한다면, 고유 코드를 사용해야 할 것이다.

만일 기존의 코드 본체에 접근하고자 한다면 고유 코드를 임시 변통 방법으로 생각하고 결국 그 코드를 자바로 이식하여야 되지 않을까? 만일 효율성에 관심이 있다면 자바 구현을 벤치마크하라. 대부분의 경우에, JIT 를 사용하는 자바의 속도는 충분할 것이다. 1996 년 Java One 컨퍼런스에서 이것을 매우 명확히 보여주었다. 선 사의 암호화 라이브러리 구현자는 그들의 암호화 함수 구현시 순수 자바가 더 적절했다고 보고를 하였다. C 구현보다는 빠르지 않다는 것이 사실이지만, 그 속도는 문제가 되지 않는 것으로 판명되었다. 자바 구현은 네트웍 I/O 보다 훨씬 빨랐다. 그리고 이 네트웍 속도가 실제적인 병목 현상임이 판명되었다.

결론적으로, 별 의미 없는 속도 증가를 위해 이식성을 희생할 필요가 없다. 다른 방법이 없을 때 최후 수단으로 고유 코드를 사용하라.

노트 : 이 장에서 우리는 자바 고유(네이티브) 인터페이스(JNI) 바인딩이라 불리는 것을 설명한다. 초기 언어 바인딩(때때로 초기 고유 인터페이스라 불림)이 자바 1.0 에서 사용되었다. 그리고 초기 바인딩의 변형이 마이크로소프트 가상 머쉰에 의해서 사용된다. 선 마이크로시스템은 여기서 설명하는 바인딩이 자바 플랫폼의 영구적인 일부이고 모든 자바 가상 머쉰에 의해서 지원될 필요가 있다는 것을 개발자들에게 보증한다.

마지막으로, 이 장에서는 고유 메소드 작성할 때 C 언어를 사용할 것이다. 왜냐하면 자바 고유 메소드로 C 가 가장 많이 사용되기 때문이다. 특히, 자바와 C 의 데이터 형식간의 대응하는 방법, 이름붙이는 방법, 함수 호출에 대해 알아볼 것이다. (이것을 보통 C 바인딩이라고 한다.)

C++노트 : 여러분은 C 대신에 C++을 사용하여 고유 메소드를 작성할 수 있다. 여기에는 몇가지 장점들이 있다-타입 검사가 보다 엄격하고 JNI 함수들에 접근하는 것이 다소 편리하다. 그러나 JNI 는 자바 플랫폼 클래스들과 C++ 클래스사이의 어떠한 직접적인 매핑도 지원하지 않는다.

자바에서 C 함수 호출하기

여러분이 좋아하는 무엇인가를 수행하는 C 함수를 가지고 있다고 가정하자. 여러분은 자바 프로그래밍 언어로 이를 다시 구현하는 하고 싶지 않다. 보다 구체적이기 위해 우리는 이 함수가 유용한 printf 함수라고 가정한다. 여러분은 여러분의 프로그램들에서 printf 를 호출하기를 원한다. 자바 프로그래밍 언어는 고유 메소드를 위해 netive 키워드를 사용한다. 그리고 클래스내에서 pintf 함수를 갭술화할 필요가 있다. 그랫 여러분은 다음과 같이 작성할 것이다:

```
public class printf
{       public native String printf(String s);
}
```

여러분은 실제 이 클래스를 컴파일할 수 있다. 그러나 여러분이 프로그램내에서 이 함수를 사용할 때 가상 머쉰은 여러분에게 printf 를 어떻게 찾는 방법을 모른다고 말할 것이다-UnsatisfiedError 를 보고한다. 따라서 방법은 해당 클래스내의 연결할 수 있는 충분한 런 타임 정보를 제공하는 것이다. 여러분이 보게 될 것처럼, JDK 하에서 이것은 3 단계

처리를 필요로 한다:

1. 자바 플랫폼 호출과 실제 C 함수사이의 전이를 수행하는 함수를 위한 C 스터브를 생성한다. 스터브는 가상 머쉰 스택의 전달인자 정보를 취함으로서 전이를 수행한다. 그리고 이를 컴파일된 C 함수에게 전달한다.
2. 특별한 공유 라이브러리를 생성하고 이 라이브러리로부터 스터브를 추출한다.
3. System.loadLinrary 라 불리는 특별한 라이브러리를 사용한다. 자바 실행 환경에게 2 단계에서 구성한 라이브러리를 로드하라고 말한다.

우리는 다양한 종류의 예제들에 대해 이러한 단계를 수행하는 방법을 보여준다. printf 의 단순한 사용에서부터 시작해서 윈도우의 레지스트리 함수(자바 플랫폼에서 직접 이용할 수 없는 명백한 플랫폼 의존 함수들)를 포함하는 실제 예제로 끝낸다.

Printf 함수와의 작업

Printf 를 사용하여 가장 단순하게 발생할 수 있는 상황으로 시작해보자: 호출되는 고유메소드는 “Hello, Native World”라는 메시지를 프린트한다. 우리는 printf 의 유용한 포맷팅 기능들을 사용하지는 않는다. 물론, 이런 코드의 이점은 없으며 단지 시작하기 좋은 방법이다. 그리고, 보다 거창한 고유 메소드를 구현하기 전에 자신의 C 컴파일러가 기대하는 대로 동작하는지 조사해 볼 수 있다.

앞서 언급했던 것처럼 여러분은 먼저, 클래스에서 고유 메소드를 선언한다. native 키워드는 자바 컴파일러에게 그 메소드가 외부에서 선언될 것임을 알린다. 물론, 고유 메소드는 자바 코드를 포함하지 않는다. 메소드 헤더는 반드시 즉시 단말 세미콜론 뒤에 따라야 한다. 따라서, 고유 메소드 선언은 추상 메소드 선언과 비슷해 보인다.

```
class HelloNative
{    public native static void greeting();
     . . .
}
```

이 특정한 예에서는 고유 메소드를 static 으로 선언하였다. 고유 메소드는 static 이거나 static 이 아닐 수 있다. 이렇게 선언한 이유는 암시적 전달인자뿐만 아니라 어떠한 전달인자 전달도 아직 다루지 않기 때문이다.

다음, 해당하는 C 함수를 작성한다. 그 함수는 반드시 자바 런타임 시스템이 기대하는

것과 정확히 똑같은 방법으로 이름을 붙여야 한다. 다음 규칙을 따라야 한다.

1. HelloNative.greeting 과 같은 완전한 자바 메소드 이름을 사용하라. 만일 HelloNative 클래스가 예를 들어 corejava 와 같은 패키지 내부에 있다면, corejava.HelloNative.greeting 을 사용하라.
2. 모든 마침표를 밑줄 문자로 변경하라. 그리고 Java_를 앞에 추가한다. 예를 들면, Java_HelloNative_greeting 또는 Java_com_horstmann_HelloNative_greeting.
3. 만일 클래스 이름이 ASCII 문자나 숫자가 아닌 것(예를 들어, '_', '$', Unicode 문자인 '\U007F' 등)을 포함하면, _0*xxxx* 로 변경하라. 여기서 *xxxx* 는 그 문자의 Unicode 문자의 16 진수 값이다. 예를 들어, 자바 이름 corejava.HelloNative.greeting 은 C 이름 corejava_HelloNative_greeting 으로 매핑된다.

노트 : 만일 여러분이 고유 메소드들을 오버로드한다면, 즉, 만일 여러분이 같은 이름을 갖는 다중 고유 메소드를 제공한다면 인코딩된 전달인자 타입으로 이중 언더스코어를 추가해야 한다. 우리는 이장의 뒤에서 전달인자 타입들의 인코딩을 설명한다. 예를 들면, 만일 여러분이 하나의 고유 메소드와, greeting, , 다른 고유 메소드, greeting(int repeat)를 가지고 있다면 첫번째 것은 Java_HelloNative_greeting__으로 호출되고 두번째 것은 Java_HelloNative_greeting__I 로 호출된다.

실제로 어느 누구도 이 작업을 직접하지는 않는다. 여러분은 자동적으로 함수 이름들을 생성해내는 javah 유틸리티를 실행할 수 있다. Javah 를 사용하기 위해, 먼저 소스 파일을 컴파일한다(예제 11-3).

```
javac HelloNative.java
```

다음으로, C 헤더 파일을 javah 유틸리티를 호출한다. javah 의 결과는 \jdk\bin 디렉토리에서 발견할 수 있다.

```
javah HelloNative
```

노트 : 만일 여러분이 여전히 JDK1.1 을 사용하고 있다면 javah 를 수행할 때 –jni 플래그를 사용해야 한다. 이 플래그가 없다면 JDK1.1 내에 javah 도구는 java1.0 바인딩을 위한 헤더 파일을 생성해낸다. JDK1.2 를 사용한다면 JNI 바인딩이 기본값이다.

---

javah 를 사용하여 예제 11-1 처럼 헤더 파일, HelloNative.h 를 생성한다.

예제 11-1 : HelloNative.h

```
/* 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:    greeting
 * Signature: ()V
 */
JNI EXPORT void JNICALL Java_HelloNative_greeting
  (JNIEnv *, jclass);

#ifdef __cplusplus
}
#endif
#endif
```

여러분이 볼 수 있듯이, 이 파일은 Java_HelloNative_greeting 함수를 선언하고 있다.(JNIEXPORT 와 JNICALL 은 헤더 파일 jni.h 에 정의되어 있다. 이들은 동적으로 로드된 라이브러리에서 제공된 함수들을 노출하기 위한 컴파일러에 의존하는 명시자들을 나타낸다.)

이제, 여러분은 헤더 파일로부터 함수 프로토타입을 소스 파일로 복사해야 한다. 그리고 예제 11-2 에서 처럼 함수의 구현 코드를 제공한다.

예제 11-2 : HelloNative.c

```
#include "HelloNative.h"
#include <stdio.h>

JNIEXPORT void JNICALL Java_HelloNative_greeting
  (JNIEnv* env, jclass cl)
{  printf("Hello world!\n");
}
```

이 간단한 함수에서 우리는 env 와 cl 전달인자를 무시한다. 여러분은 나중에 이들을 보게될 것이다.

C++노트 : 여러분은 고유 메소드를 작성하기 위해 C++을 사용할 수 있다. 그러나 여러

분은 extern “C”로서 고유 메소드들을 구현하는 함수 선언을 해야 한다. 예를 들면,

```
#include "HelloNative.h"
#include <stdio.h>

extern “C”
JNIEXPORT void JNICALL Java_HelloNative_greeting
  (JNIEnv* env, jclass cl)
{  printf("Hello world!\n");
}
```

다음으로 C 코드를 동적으로 로드되는 라이브러리로 컴파일한다. 예를 들면, 윈도우에서 마이크로소프트 C++ 컴파일러에서 사용하는 명령어는 다음과 같다.

cl –Ic:\jdk\include –Ic: :\jdk\include\win32 –LD HelloNative.c –FeHelloNative.dll

솔라리스 운영체제하의 선 컴파일러에서는 다음의 명령어를 사용한다.

cc –G –I/usr/local/jdk/include/ -I/usr/local/jdk/include/solaris HelloNative.c –o libHelloNative.so

(여러분은 헤더 파일들의 위치를 명시하기 위해 JDK 를 포함하고 있는 경로에 따라 다른 경로를 사용할 필요가 있다.)

팁 : 만일 여러분이 마이크로소프트 컴파일러를 사용하여 DOS 쉘하에서 DLL 을 컴파일한다면 먼저 배치 파일을 수행한다.

C:\devstudio\vc\bin\vcvars32.bat

이 배치 파일은 컴파일러가 필요로 하는 환경변수와 경로를 설정함으로서 명령어 라인 컴파일러를 적절히 조정한다.

마지막으로, 우리는 가상 머신이 먼저 클래스를 사용하기 전에 라이브러리를 로드하도록 System.loadLibrary 메소드 호출을 추가할 필요가 있다. 이를 위한 가장 쉬운 방법은 예제 11-3 에서 처럼 고유 메소드를 포함하고 있는 클래스내에서 정적 초기화 블록을 사용하는 것이다.

예제 11-3: HelloNative.java

```
class HelloNative
{  public static native void greeting();
   static
   {  System.loadLibrary("HelloNative");
   }
}
```

여러분이 앞서 주어진 모든 과정을 따랐다고 가정하면 예제 11-4 에서 보여지는 HelloNativeTest 응용 프로그램을 수행할 준비가 되어있다.

예제 11-4: HelloNativeTest.java

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

만일 여러분이 이 프로그램을 컴파일하고 수행하면 터미날 윈도우상에 "Hello,Native World!" 메제지가 출력된다.

물론, 그 자체로 특별한 인상을 주는 것은 아니다. 그러나 만일 여러분이 C printf 명령어에 의해서 생성된 해당 메시지를 기억하고 있지 않다면 두개 언어사이의 차이를 연결하기위해 우리가 수행한 첫번째 단계를 찾게될 것이다.

노트: 고유 코드를 위한 몇몇 공유 라이브러리들은 초기화 코드를 수행할 필요가 있다. 여러분은 모든 초기화 코드를 JNI_OnLoad 메소드내에 위치시킬 수 있다. 유사하게, VM 이 셧 다운 될 때 만일 여러분이 이를 제공한다면 JNI_OnUnload 메소드를 호출할 것이다. 프로토타입은 다음과 같다.

jint JNI_OnLoad(JavaVM* void* reserved)

void JNI_OnUnload(javaVM* vm , void* reserved)

JNI_OnLoad 메소드는 JNI_VERSION_1_1 처럼 필요로 하는 최소의 VM 버전을 반환할 필요가 있다.

---

java.lang.System

- void loadLibrary(String libname)

주어진 이름을 라이브러리를 로드한다. 그 라이브러리는 라이브러리 검색 경로에 있어야 한다. 라이브러리를 찾는 정확한 방법은 운영 체제에 따라 다를 수 있다. DOS/Windows 의 경우, 우선 현재 디렉토리에서 찾고 그 다음에는 PATH 에 있는 디렉토리가 검색된다.

- void load(String filename)
  주어진 파일 이름을 가진 라이브러리를 로드한다. 만일 라이브러리가 발견되지 않으면, 자바는 UnsatisfiedLinkError 를 던진다.

숫자 전달인자와 반환값

C 와 자바 코드 사이에 숫자를 전달할 때, C 의 데이터 형에 대응하는 자바의 데이터 형을 이해하는 것이 중요하다. 가령, C 는 int 와 long 데이터 형을 가지고 있지만 그 구현은 하부의 플랫폼에 따라 다르다. 어떤 플랫폼에서는 int 가 16 비트를 차지하는 반면에 다른 시스템에서는 32 비트를 차지한다. 물론, 자바에서는 int 는 항상 32 비트이다. 이러한 이유에서 자바 고유 인터페이스는 jint,jlong 같은 타입을 정의한다.

다음 표 11-1 은 자바와 C 데이터 형 사이의 대응이다.

표 11-1: 자바 데이터 타입과 C 타입

Java	C	바이트
boolean	jboolean	1
byte	jbyte	1
char	char	2
short	jshort	2
int	jint	4
long	jlong	8
float	jfloat	4
double	jdouble	8

헤더파일 jni.h 에서 이들 타입들은 목표 플랫폼상에 동등한 타입으로서 typedef 구문으로 선언된다. 또한 헤더 파일은 JNI_FLASE=0 과 JNI_TRUE=1 상수를 정의한다.

숫자를 포맷팅하기 위해 printf 사용하기

자바 라이브러리는 부동 소수들을 포맷하여 출력하기 위한 멋진 방법을 갖지 못하다는 사실을 상기하자. 물론 여러분은 (1 권, 3 장에서 살펴본) DecimalFormat 클래스와 사용자 포맷을 사용할 수 있다-우리는 printf 를 간단히 호출하는 것만큼 쉽다고는 생각하지 않는다. printf 는 빠르고 쉽고 잘알려져 있기 때문에 고유 메소드에서 printf 함수를 호출하여 같은 기능을 구현하기로 결정했다고 가정하자.

우리는 실제 이 접근방법을 추천하는 것은 아니다: 고유 코드는 모든 목표 플랫폼에 대해 컴파일할 필요가 있다. 우리는 고유 메소드에 전달인자를 넘겨주고 반환 값을 얻는 기술을 보여주지 않았기 때문에 이를 사용한다.

예제 11-5 는 주어진 필드 폭과 정밀도를 갖는 부동소수를 출력하기 위해 고유 메소드를 사용하는 Printf1 을 호출하는 클래스를 보여준다.

예제 11-5: Printf1Test.java

```
class Printf1
{  public static native int print(int width, int precision, double x);
   static
   { System.loadLibrary("Printf1");
   }
}
```

C 로 메소드로 구현할 때 모든 int 와 double 전달인자는 예제 11-6 에서 처럼 jint 와 jdouble 로 변환된다.

예제 11-6: Printf1.c

```
#include "Printf1.h"
#include <stdio.h>

JNIEXPORT jint JNICALL Java_Printf1_print
  (JNIEnv* env, jclass cl, jint width, jint precision, jdouble x)
{  char fmt[30];
   jint ret;
   sprintf(fmt, "%%%d.%df", width, precision);
   ret = printf(fmt, x);
   return ret;
}
```

함수는 단순하게 변수 fmt 내에서 "%w.pf" 포맷 문자열을 모으고 printf 를 호출한다. 그런 다음 출력한 문자들의 수를 반환한다.

예제 11-7 은 Printf1 클래스의 데모를 보여주는 테스트 프로그램이다.

예제 11-7: Printf1Test.java

```
class Printf1Test
{  public static void main(String[] args)
   {  int count = Printf1.print(8, 4, 3.14);
      count += Printf1.print(8, 4, (double)count);
      System.out.println();
      for (int i = 0; i < count; i++)
         System.out.print("-");
      System.out.println();
   }
}
```

문자열 전달인자

다음에는, 문자열을 고유 메소드에 전달하거나 받는 방법을 생각해 보자. 자바 프로그래밍 언어에서 문자열은 16 비트 Unicode 문자들의 연속이며 C 문자열은 8 비트 문자들의 null 로 끝나는 문자열이다. 따라서 두 문자열은 아주 다르다. 자바 고유 인터페이스는 두개의 함수 셋을 가지고 있다. 하나는 자바 문자열을 UTF(Unicode Text Format)으로 변경하고 하나는 자바 문자열을 유니코드 문자들의 배열로 변환한다. 즉, jchar 배열로 변환한다. UTF 포맷은 1 장에서 설명했었다.-ASCII 문자들은 있는 그대로 인코딩되지만 모든 유니코드 문자들은 연속되는 2 바이트나 3 바이트로 인코딩된다.

만일 여러분의 C 코드가 이미 유니코드를 사용한다면 여러분은 두번째 변환 함수들의 셋을 사용하길 원하게 될 것이다. 다른 한편으로 만약에 여러분의 모든 자바 문자열들이 ASCII 문자들로 제약된다면 UTF 변환 함수들을 사용할 수 있다.

String 전달인자를 가진 고유 메소드는 실제로 jstring 형의 구조체에 대한 포인터를 받는다. String 형의 값을 반환하는 자바 메소드는 반드시 jstring 구조체에 대한 포인터를 반환해야 한다. 예를 들면, NewStringUTF 함수는 UTF 인코딩된 문자들을 포함하고 있는 char 배열로부터 새로운 jstring 객체를 만든다. 불운하게도 JNI 함수들은 다소 이상한 호출 방식을 가지고 있다. 다음은 NewStringUTF 함수에 대한 호출이다.

```
JNIEXPORT jstring JNICALL Java_HelloNative_getGreeting(JNIEnv* env, jclass cl)
{           jstring jstr;
            char greetingp[] = “Hello, Native World\n”;
            jstr = (*env)->NewStringUTF(env,greeting);
```

```
    return jstr;
}
```

JNI 함수에 대한 모든 호출은 모든 고유 메소드의 첫번째 전달인자인 env 포인터를 사용한다. env 포인터는 함수 포인터들의 테이블에 대한 포인터이다.(그림 11-1 참조) 그러므로 여러분은 모든 JNI 호출시 그 앞에 실제로 함수 포인트를 실제 참조해주는 (*env)->를 첨가해야 한다. 이러한 설정은 다소 번거로워 보인다. 그리고 C 프로그래머들에게는 보다 투명해보일 것이다. 우리는 여러분에게 함수 포인터들의 테이블에 대해 걱정할 필요없이 (*env)->를 제공하길 제안한다.

그림 11-1 : env 포인터

C++노트 : C++에서 JNI 함수들을 접근하는 것이 보다 간단하다. JNIEnv 클래스의 C++버전은 여러분을 위해 함수 포인터를 조회하게 해주는 인라인 멤버 함수를 가지고 있다.
예를 들면, 여러분은 NewStringUTF 함수를 다음처럼 호출할 수 있다.

```
jstr = env->NewStringUTF(greeting);
```

여러분이 호출의 전달인자 목록에서 JNIEnv 포인터를 생략한다는 것을 기억하라.

NewStringUTF 함수는 새로운 jstring 을 생성하게 해준다. 존재하는 jstring 객체의 내용을 읽기위해, GetStringUTFChars 함수를 사용하라. 이 함수는 문자열을 묘사하는 UTF 문자들에 대한 const jbyte* 포인터를 반환한다. 특정 가상 머쉰은 내부 문자열 표현을 위해 UTF 를 사용할 의무는 없다. 그래서 여러분은 실제 자바 문자열에 대한 문자 포인터를 얻어야 할 것이다. 자바 문자열들은 변하지않기 때문에 심각하게 const 를 다루고 해당 문자 배열내에 쓰기 시도를 하지 말아야 한다는 점이 아주 중요하다. 다른 한편으로, 만일 가상머쉰이 내부 문자열 표현을 위해 유니코드 문자들을 사용한다면 해당 함수 호출은 동일한 UTF 로 채워질 새로운 메모리 블록을 할당한다.

가상 머쉰은 여러분이 UTF 문자열의 사용을 마쳤을 때를 알아야 한다. 마쳤을 때를 알아야 가베지를 수집을 할 수 있다.(가베지 컬렉터는 별도의 쓰레드로 수행된다, 그리고 고유 메소드들의 실행을 중단할 수 있다.) 이러한 이유에서, 여러분은 ReleaseStringUTFChars 함수를 호출해야 한다. 대안으로, 여러분은 GetStringRegion 이나

GetStringUTFRegion 메소드를 사용하여 문자열의 문자들을 유지하기 위한 여러분 자신의 버퍼를 제공할 수 있다.

마지막으로, GetStringUTFLength 함수는 문자열의 UTF 인코딩을 위해 필요한 문자들의 수를 반환한다.

C 코드에서 자바 문자열 접근하기

- jstring NewStringUTF(JNIEnv* env, const char bytes[])
  UTF 문자열로부터 새로운 자바 문자열을 반환하거나 만약 문자열이 생성되지 않으면 NULL을 반환한다.
  전달인자: env JNI 인터페이스 포인터
  bytes 널로 종료되는 UTF 문자열

- jsize GetStringUTFLength(JNIEnv* env, jstring string)
  UTF 인코딩시 필요한 문자들의 수를 반환한다.
  전달인자: env JNI 인터페이스 포인터
  string 자바 문자열 객체

- const jbyte* GetStringUTFChars(JNIEnv* env, jstring string, jboolean* isCopy)
  문자열의 UTF 인코딩에 대한 포 string 포인터를 반환하거나 문자 배열이 생성되지 않으면 NULL 을 반환한다. 포인터는 ReleaseStringUTFChars 를 호출할 때까지 유효하다.
  전달인자: env JNI 인터페이스 포인터
  string 자바 문자열 객체
  isCopy 만일 복사가 되면 JNI_TRUE로 채워진 jboolean에 대한 포인터, 그렇지 않으면 JNI_FALSE로 채워진다.

- void ReleaseString UTFChars(JNIEnv* env, jstring string, const jbyte bytes[])
  가상 머쉰에게 고유 코드가 더 이상 bytes 를 통해서 자바 문자열에 대한 접근을 필요로 하지 않는다는 것을 알린다.
  전달인자: env JNI 인터페이스 포인터
  string 자바 문자열 객체
  bytes GetStringUTFChars 에 의해서 반환되는 포인터

- void GetStringRegion(JNIEnv* env, jstring string, jsize start, jsize length, jchar *buffer)

문자열로부터 일련의 유니코드 문자들을 사용자가 제공한 버퍼에 복사한다.

전달인자: env JNI 인터페이스 포인터
string 자바 문자열 객체
start 시작 인덱스
length 복사할 문자들의 수
buffer 사용자가 제공한 버퍼

- void GetStringUTFRegion(JNIEnv* env, jstring string, jsize start, jsize length, jchar *buffer)
  문자열로부터 일련의 UTF8 문자들을 사용자가 제공한 버퍼에 복사한다. 버퍼는 바이트들을 충분히 유지할 수 있도록 길어야 한다. 최악의 경우, 3 X length 바이트들이 복사된다.

- jstring NewString(JNIEnv* env, const jchar chars[], jsize length)
  유니코드 문자열로부터 새로운 자바 문자열 객체를 반환하거나 문자열이 생성되지 못하면 NULL을 반환한다.

  전달인자: env JNI 인터페이스 포인터
  chars 널로 종료되는 UTF 문자열
  length 문자열내에 문자 수

- jsize GetStringLength(JNIEnv* env, jstring string)
  문자열내의 문자들의 수를 반환한다.

  전달인자: env JNI 인터페이스 포인터
  string 자바 문자열 객체

- const jchar* GetStringChars(JNIEnv* env, jstring string, jboolean* isCopy)
  문자열의 유니코드 인코딩에 대한 포인터를 반환하거나 문자 배열이 생성되지 못하면 NULL을 반환한다. 포인터는 ReleaseStringChars가 호출될 때까지 유효하다.

  전달인자: env JNI 인터페이스 포인터
  string 자바 문자열 객체
  isCopy 만일 복사가 되면 JNI_TRUE로 채워진 jboolean에 대한 포인터, 그렇지 않으면 JNI_FALSE로 채워진다.

- void ReleaseStringChars(JNIEnv* env, jstring string, const jchar chars[])
  가상 머쉰에게 고유 코드가 더 이상 chars를 통해서 자바 문자열에 대한 접근을 필요로 하지 않는다는 것을 알린다.

  전달인자: env JNI 인터페이스 포인터

string 자바 문자열 객체

chars GetStringChars 에 의해서 반환되는 포인터

고유 메소드에서 sprintf 호출하기

C 함수의 sprintf 와 동등한 자바 코드를 작성하기 위해 위 함수들을 사용해 보자. 예제 11-8 과 같이 함수들을 호출할 것이다.

예제 11-8: Printf2Test.java

```
class Printf2Test
{  public static void main(String[] args)
   {   double price = 44.95;
       double tax = 7.75;
       double amountDue = price * (1 + tax / 100);

      String s = Printf2.sprintDouble("Amount due = %8.2f", amountDue);
       System.out.println(s);
   }
}
```

예제 11-9 는 고유 sprintf 메소드를 갖는 클래스를 보여준다.

예제 11-9:Printf2.java

```
class Printf2
{   public static native String sprint(String format, double x);
    static
    {   System.loadLibrary("Printf2");
    }
}
```

그러므로, 부동소수점 숫자를 포맷하는 C 함수는 다음과 같은 프로토타입을 갖는다.

```
JNIEXPORT jstring JNICALL Java_Printf2_sprint
   ( JNIEnv* env, jclass cl, jstring format, jdouble x)
```

예제 11-10 은 C 구현을 위한 코드를 보여준다. GetStringUTFChars 에 대한 호출은 포맷 전달인자를 읽는다. NewStringUTF 은 반환값을 생성한다. 그리고 ReleaseStringUTFChars 는 가상 머쉰에게 문자열에 대한 접근이 더 이상 필요하지 않다는 것을 알린다.

예제 11-10 : Printf2.c

```
#include "Printf2.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

char* find_format(const char format[])
/**
 * @param format a string containing a printf format specifier
 * (such as "%8.2f"). Substrings "%%" are skipped.
 * @return a pointer to the format specifier (skipping the '%')
 * or NULL if there wasn't a unique format specifier
 */
{  char* p;
   char* q;

   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
   /* now check that % is unique */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating point format */
   return p;
}

JNIEXPORT jstring JNICALL Java_Printf2_sprint
  (JNIEnv* env, jclass cl, jstring format, jdouble x)
{  const char* cformat;
   char* fmt;
   jstring ret;

   cformat = (*env)->GetStringUTFChars(env, format, NULL);
   fmt = find_format(cformat);
   if (fmt == NULL)
      ret = format;
   else
   {  char* cret;
      int width = atoi(fmt);
      if (width == 0) width = DBL_DIG + 10;
      cret = (char*)malloc(strlen(cformat) + width);
      sprintf(cret, cformat, x);
```

```
      ret = (*env)->NewStringUTF(env, cret);
      free(cret);
   }
   (*env)->ReleaseStringUTFChars(env, format, cformat);
   return ret;
}
```

이 함수에서, 오류 처리를 매우 간단히 하였다. 만일 부동소수점 숫자를 프린트하는 포맷 코드가 *%w.pc* (*c* 는 e, E, f, g, G 중의 하나) 형태가 아니라면, 단순히 그 숫자를 포맷하지 않는다. 나중에 고유 메소드가 예외를 던지는 방법에 대해서 알아볼 것이다.

객체 필드에 접근하기

지금까지 본 모든 고유 메소드는 숫자나 문자열 전달인자를 가진 정적 메소드였다. 다음에는 객체에 대해 연산하는 고유 메소드를 생각해 보자. 연습으로, 우리는 1 권의 4 장에서 소개한 Employee 클래스의 메소드를 고유 메소드를 사용하여 구현할 것이다. 이것은 일반적으로 여러분이 하고자 하는 것이 아니다. 그러나 고유 메소드로부터 객체 필드들에 접근하는 방법의 예제를 보여줄 것이다.

raiseSalary 메소드를 고려해보자. 자바 프로그래밍 언어에서 코드는 단순하다.

```
public void raiseSalary(double byPercent)
{       salary *= 1+byPercent / 100;
}
```

이 메소드를 고유 메소드로 다시 작성해보자. 앞선 고유 메소드의 예제와 달리 이 메소드는 정적 메소드가 아니다. 다음의 프로토타입을 제공하여 javah 를 구동한다.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary
        (JNIEnv *, jobject, jdouble)
```

두번째 전달인자에 주목하자. 더 이상 jclass 타입이 아니라 jobject 타입이다. 사실, 이것은 this 참조와 동일하다. 정적 메소드들은 클래스에 대한 참조를 얻는다. 반면에 비정적 메소드는 암묵적인 this 전달인자에 대한 참조를 얻는다.

이제 우리는 암묵적인 전달인자의 salary 필드에 접근한다. 자바 1.0 의 자바-C 바인딩에서 이것은 쉽다-프로그래머는 직접 객체 데이터 필드들에 접근할 수 있다. 그러나 직접 접근은 모든 가상 머쉰들이 그들의 내부 데이터 레이아웃을 노출할 것을 요구한다. 이러

한 이유 때문에, JNI 는 프로그래머들이 특별한 JNI 함수들을 호출함으로써 데이터 필드들의 값들을 설정하거나 얻어올 것을 요구한다.

우리의 경우, salary 가 double 티입이기 때문에 GetDoubleField 와 SetDoubleField 함수들을 사용할 필요가 있다. 다른 필드 타입들을 위한 함수들이 존재한다- GetIntField/SetIntField,GetObjectField/SetObjectField 등등. 일반적인 구문은 다음과 같다:

```
x = (*env)->GetXxxField(env, class, fieldID);
(*env)->GetXxxField(env,class,fielded,x);
```

여기서, class 는 Class 타입의 자바 객체를 표현하는 값이다. 그리고 fieldID 는 특별한 값이다. JfieldID 는 구조체내의 필드를 실별하고 Xxx 는 자바 데이터 타입(Object,Boolean,Byte,기타 등등)을 표현한다. class 객체를 얻는데에는 2 개의 방법이 있다. GetObjectClass 함수는 모든 객체의 클래스를 반환한다. 예를 들면:

```
jclass class_Employee = (*env)->GetObjectClass(env,obj_this);
```

FindClass 함수는 여러분에게 문자열로 클래스 이름을 명시하게 해준다(주의할 것은 패키지 이름 구별자로 피리어드대신 /를 사용한다는 점이다).

```
jclass class_String
        = (*env)->FindClass(env,"java/lang/String");
```

fieldID 를 얻기위해 GetFieldID 함수를 사용한다. 여러분은 필드의 이름과 서명, 타입의 인코딩을 제공해야 한다. 가령, 다음은 salary 필드의 필드 ID 를 얻기위한 코드이다.

```
jfieldID id_salary
        = (*env)->GetFieldID(env,class_Employee,"salary","D");
```

문자열 "D"는 double 타입을 나타낸다. 여러분은 다음 절에서 서명을 인코딩하기 위한 완전한 규칙을 배우게 될 것이다.

여러분은 데이터 필드에 접근하는 것이 아주 얽혀있는 것처럼 생각할 수도 있다. 그러나 JNI 의 디자이너들은 데이터 필드들을 직접 노출하는 것을 원하지 않기 때문에 필드 값들을 설정하고 가져오기 위한 함수들을 제공한다. 이러한 함수들의 비용을 최소화하기 위해 필드 이름으로부터의 필드 ID 계산을 별도의 단계로 설정하였다. 즉, 만일 여러분이 특정 필드의 값을 반복적으로 설정하거나 가져온다면 필드 식별자를 계산하기위한 비용을 오직 한번 초래한다.

모든 조각들을 한곳에 모아보자. 다음 코드는 고유 메소드로 raiseSalary 메소드를 다시 구현한 것이다.

```
JNIEXPORT void JNICALL Java_Employee_raiseSalary
        (JNIEnv * env, jobject obj_this,jdouble byPercent)
{       /* get the class */
        jclass class_Employee = (*env)->GetObjectClass(env,obj_this);

        /* get the field ID */
        jfieldID id_salary = (*env)->GetFieldID(env,class_Employee, "salary", "D");

        /* get the field value */
        jdouble salary = (*env)->GetDoubleField(env,obj_this,id_salary);

        salary *= 1 + byPercent / 100;

        /* set the field value */
        (*env)->SetDoubleField(env, obj_this, id_salary, salary);
}
```

주의 : 클래스 참조는 오직 고유 메소드가 반환할 때까지 유효하다. 따라서 여러분은 코드에서 GetObjectClass **의 반환값을 감지할 수 없다. 나중에 메소드 호출에서 재사용을 위해 클래스 참조를 저장하지 않는다. 여러분은 고유 메소드가 실행될 때마다** GetObjectClass **를 호출할 필요가 있다. 만일 이것이 견디기 힘들다면** NewGlobalRef **를 사용하여 참조를 잠궈둘 수 있다**:

```
static jclass class_x = 0;
static jfieldID id_a;
…
if(class_x ==0)
{       jaclass cx = (*env)->GetObjectClass(env,obj);
        class_X =  (*env)-> NewGlobalRef(env,cx);
        id_a = (*env)->GetFieldID(env, clas, "a","…");
}
```

이제 여러분은 연속적인 호출로 클래스 참조와 필드 ID **를 사용할 수 있다. 클래스를 사용하여 작업을 마칠 때, 다음 호출을 해야한다.**

```
(*env)-> DeleteGlobalRef(env,class_x)
```

예제 11-11 과 11-12 는 테스트 프로그램과 Employee 클래스를 위한 자바 코드를 보여준다. 예제 11-13 은 고유 raiseSalary 메소드를 위한 C 코드를 포함한다.

정적 필드들에 접근하기

정적 필드들에 접근하는 것은 비정적 필드들에 접근하는 것과 유사하다. 여러분은 GetStaticFieldID 와 GetStaticXxxFieldID/ SetStaticXxxFieldID 함수를 사용한다. 이들 메소드는 거의 비정적 파트에 동일하게 대응한다. 오직 두개의 차이가 존재한다.

- 여러분은 객체를 갖지않기 때문에, 클래스 참조를 얻기위해 GetObjectClass 대신에 FindClass 를 사용해야 한다.
- 여러분은 필드에 접근할 때 인스턴스 객체가 아니라 클래스를 제공한다.

예를 들면, 다음은 System.out 에 대한 참조를 얻을 수 있는 방법이다.

```
/* get the class */
jclass class_System = (*env)->FindClass(env,"java/lang/system");

/* get the field ID */
jfieldID id_out = (*env)->GetStaticFieldID(env,class_System, "out",
        "Ljava/io/PrintStream;");

/* get the field value */
jobject obj_out = (*env)->GetStaticObjectField(env,class_System,id_out);
```

예제 11-11:EmployeeTest.java

```
public class EmployeeTest
{  public static void main(String[] args)
   {  Employee[] staff = new Employee[3];

      staff[0] = new Employee("Harry Hacker", 35000);
      staff[1] = new Employee("Carl Cracker", 75000);
      staff[2] = new Employee("Tony Tester", 38000);

      int i;
      for (i = 0; i < 3; i++) staff[i].raiseSalary(5);
      for (i = 0; i < 3; i++) staff[i].print();
   }
}
```

예제 11-12:Employee.java

```
public class Employee
{  public Employee(String n, double s)
   {  name = n;
      salary = s;
   }

   public native void raiseSalary(double byPercent);

   public void print()
   {  System.out.println(name + " " + salary);
   }

   private String name;
   private double salary;

   static
   {  System.loadLibrary("Employee");
   }
}
```

예제 11-13:Employee.c

```
#include "Employee.h"

#include <stdio.h>

JNIEXPORT void JNICALL Java_Employee_raiseSalary
  (JNIEnv* env, jobject obj_this, jdouble byPercent)
{  /* get the class */
   jclass class_Employee = (*env)->GetObjectClass(env,
      obj_this);

   /* get the field ID */
   jfieldID id_salary = (*env)->GetFieldID(env,
      class_Employee, "salary", "D");

   /* get the field value */
   jdouble salary = (*env)->GetDoubleField(env, obj_this,
      id_salary);

   salary *= 1 + byPercent / 100;

   /* set the field value */
   (*env)->SetDoubleField(env, obj_this, id_salary, salary);
}
```

객체 필드들에 접근하기

- jfieldID GetFieldID(JNIEnv *env, jclass cl, const char name[], const sig[])

  클래스내의 필드의 식별자를 반환한다.

  전달인자: env JNI **인터페이스 포인터**

cl	클래스 객체
name	필드 이름
sig	인코딩된 필드 서명

- Xxx GetXxxField(JNIEnv *env, jobject obj, jfieldId id)

  필드 값을 반환한다. 필드 타입 Xxx는 Object, Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다.

  전달인자:

env	JNI 인터페이스 포인터
obj	반환된 필드를 갖는 객체
id	필드 식별자

- void SetXxxField(JNIEnv *env,jobject obj, jfieldID id, Xxx value)

  필드에 새로운 값을 설정한다. 필드 타입 Xxx는 Object, Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다.

  전달인자:

env	JNI 인터페이스 포인터
obj	설정한 필드를 갖는 객체
id	필드 식별자
value	새로운 필드 값

- jfieldID GetStaticFieldID(JNIEnv *env,jclass cl, const char name[], const char sig[])

  클래스내의 정적 필드의 식별자를 반환한다.

  전달인자:

env	JNI 인터페이스 포인터
cl	클래스 객체
name	필드 이름
sig	인코딩된 필드 서명

- Xxx GetStaticXxxField(JNIEnv *env,jclass cl,jfieldID id)

  정적 필드의 값을 반환한다. 필드 타입 Xxx는 Object, Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다.

  전달인자:

env	JNI 인터페이스 포인터
cl	설정된 정적 필드를 갖는 클래스 객체
id	필드 식별자

- void SetStaticXxxField(JNIEnv *env,,jclass cl,jfieldID id, Xxx value)

  정적 필드에 새로운 값을 설정한다. 필드 타입 Xxx 는 Object, Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다.

전달인자:	env	JNI 인터페이스 포인터
	cl	설정된 정적 필드를 갖는 클래스 객체
	id	필드 식별자
	value	새로운 필드 값

서명(Signature)

자바 프로그래밍 언어에 정의된 메소드들을 호출하고 객체 필드에 접근하기 위해서, 여러분은 데이터 타입과 메소드 서명의 이름을 만드는 규칙을 알아야 한다. (메소드 서명은 전달인자와 그 메소드의 반환형을 설명한다.) 다음은 그 인코딩 일람표이다.

B	byte
C	char
D	double
F	float
I	int
J	long
L*classname;*	클래스형
S	short
V	void
Z	boolean

L 표현의 끝에 있에 있는 세미콜론은 전달인자들 사이의 분리가 아니라 타입 표현의 종료를 말한다. 예를 들어, 다음 생성자는

Employee(java.lang.String, double, java.util.Date)

서명은 다음과 같다.

"(Ljava/lang/String;Dljava/util/Date;)V"

여러분이 볼 수 있듯이, D 와 Ljava/util/Date; 사이에는 구분이 없다.

또한 이러한 인코딩 일람표에서 여러분은 패키지와 클래스 이름들을 구별하기 위해 . 대신에 /를 사용해야 한다는 것을 기억하라.

배열 타입을 설명하려면, [을 사용하라. 예를 들어, 문자열 배열은 다음과 같다.

[Ljava/langString;

float[][]은 다음과 같다.

[[F

메소드의 완전한 서명을 위해, 여러분은 괄호쌍내에 전달인자 타입들을 나열한 다음 반환 타입을 나열한다. 예를 들면, 두개의 정수를 받고 정수를 반환하는 메소드는 다음처럼 인코딩된다.

(II)I

앞절의 예제에서 사용했던 출력 메소드는 다음의 서명을 갖는다.

(Ljava/lang/String;)V

즉, 메소드는 문자열을 받고 void 를 반환한다.

팁 : 여러분은 클래스 파일들로부터 필드 서명들을 생성해내기 위해 –s 옵션으로 javap 명령어를 사용할 수 있다. 예를 들면:

javap –s –private Classname

여러분은 모든 필드들과 메소드들의 서명을 출력하는 다음의 결과를 얻는다.

```
public synchronized class Employee extends java.lang.Object
        /* ACC_SUPER bit set */
{
        private java.lang.String name;
        /* Ljava/lang/String */
        private double salary;
        /* D */
        private java.util.Date hireDay;
        /* Ljava/util/Date */
        public Employee(java.lang.String, double, java.util.Date);
```

```
        /*  (Ljava/lang/String;Dljava/util/Date;)V   */
        public void print();
        /* ()V */
        public void raiseSalary(double);
        /* ()V */
        public int hireYear();
        /* ()I */
    }
```

노트: 서명을 설명할 때 이 만들기 일람표를 강제로 사용해야만 하는 합리적인 이유는 없다. 자바의 설계자는 void(int, java.lang.String)와 같은 자바 스타일의 서명으로 쉽게 함수가 작성될 수 있도록 하며 그들이 원하는 내부 표현으로 인코드할 수도 있다. 그리고 다시, 만들어진 서명은 프로그래밍을 자바 가상 기계에 가깝도록 신비감을 가져다 준다.

자바 메소드 호출하기

물론, 자바 함수는 C 함수를 호출할 수 있다. 그것이 바로 고유 메소드가 의미하는 바이다. 다른 방법으로 C에서 자바 코드를 호출하는 방법이 있을까? 왜 이러한 방법이 필요할까? 그 대답은 고유 메소드가 자신에게 전달된 객체로부터 서비스를 요청할 필요가 가끔씩 있다는 것이다. 우리는 먼저 여러분에게 비정적 메소드에 대해 이러한 작업을 수행하는 방법을 보여준 다음 정적 메소드에 대한 방법을 보여준다.

비정적 메소드

고유 코드에서 자바 메소드를 호출하는 예로써, Printf 클래스를 개선하여 C 함수인 fprintf와 비슷하게 동작하도록 멤버 함수를 추가하고 Printf 클래스를 강화해보자. 즉, 문자열을 임의의 PrintWriter 객체에 프린트할 수 있도록 하는 것이다.

```
class Printf3
{  public static native void fprintDouble(PrintWriter out,
       String s, double x);
   . . .
}
```

우선 이미 구현된 sprint 메소드에 정의된 것과 같이, String 객체 str 에 프린트되도록 문자열을 모은다. 그리고 고유 메소드를 구현한 C 함수로부터 PrintWriter 클래스의 print 메소드를 호출한다.

다음 함수 호출을 사용하여 C 에서 어떤 자바 메소드라도 호출할 수 있다.

```
(*env)->CallXxxMethod(env,implicit parameter, methodID, explicit parameters)
```

Xxx 를 Void,Int,Object 처럼 메소드의 반환 타입에 따라 대체한다. 객체의 필드에 접근하기 위해 fieldID 가 필요한 것처럼 여러분은 메소드를 호출하기 위해 메소드 ID 가 필요하다. 여러분은 JNI 함수 GetMethodID 를 호출하여 메소드 ID 를 얻을 수 있다. 그리고 클래스, 메소드 이름, 메소드 서명을 제공한다.

예제에서, 우리는 PrintWriter 클래스의 print 메소드의 ID 를 얻길 원하다. 1 장에서 본 것처럼 PrintWriter 클래스는 9 개 다른 메소드들을 가지고 있고 모두 print 를 호출한다. 이러한 이유에서, 여러분은 또한 사용하길 원하는 특별한 함수의 반환값과 전달인자들을 표현한 문자열을 제공해야 한다. 예를 들면, 우리는 void print(java.lang.String)을 사용하길 원한다. 앞선 절에서 설명한 것처럼, 이제 우리는 서명을 “Ljava/lang/string;)V” 문자열로 표현해야 한다.

다음은 메소드 호출을 만들기 위한 완전한 코드이다:

1. 묵시적인 전달인자의 클래스를 얻는다;
2. 메소드 ID 를 얻는다.
3. 호출을 만든다.

```
/* get the class */
class_PrintWriter =(*env)->GetObjectClass(env, out);

/* get the method ID */
id_print = (*env)->GetMethodID(env, class_PrintWriter, “print”, “(Ljava/lang/String;)V”);

/* call the method */
(*env)->CallVoidMethod(env, out, id_print, str);
```

예제 11-14 와 11-15 는 테스트 프로그램을 위한 자바 코드와 Printf3 클래스를 보여준다. 예제 11-16 은 고유 fprint 메소드를 위한 C 코드를 포함하고 있다.

노트 : 숫자 메소드 ID 와 필드 ID 는 리프렉션 API 내의 Method 와 Field 객체와 개념적으로 유사하다. 여러분은 다음의 함수들을 사용하여 이들 간의 변환을 할 수 있다:

```
jobject ToReflectedMethod(JNIEnv* env, jclass class, jmethodID methodID); // return Method object
methodID FromReflectedMethod(JNIEnv* env, jobject method);
jobject ToReflectedField(JNIEnv* env, jclass class, jfieldID fieldID); // returns field object
fieldID FromReflectedField(JNIEnv* env, jobject field);
```

정적 메소드

고유 메소드에서 정적 메소드를 호출하는 것은 비정적 메소드를 호출하는 것과 유사하다. 다음의 2 가지 차이점이 있다.

- GetStaticMethodID 와 CallStaticXxxMethod 함수를 사용한다.
- 메소드를 호출할 때, 암묵적인 전달인자 객체가 아니라 클래스 객체를 제공한다.

예제로, 고유 메소드에서 정적 메소드에 대한 호출을 만들어 보자.

```
System.getProperty("java.class.path")
```

이 호출의 반환값은 현재 클래스 경로를 제공하는 문자열이다.

먼저, 우리는 사용할 클래스를 찾을 필요가 있다. 쉽게 이용할 수 있는 System 클래스의 객체를 갖고 있지 않기 때문에 GetObjectClass 보다 FindClass 를 사용한다.

```
jclass class_System = (*env)->FindClass(env, "java/lang/System");
```

다음, 우리는 getProperty 정적 메소드의 ID 를 필요로 한다. 메소드의 인코딩된 서명은 다음과 같다.

```
"(Ljava/lang/String;)Ljava/lang/String;"
```

전달인자와 반환값 모두가 문자열이기 때문이다. 따라서 우리는 다음과 같은 메소드 ID 를 얻을 수 있다.

```
jmethodID Property =(*env)->GetStaticMethodID(env,class_system,"getProperty",
        "(Ljava/lang/String;)Ljava/lang/String;);
```

마지막으로, 우리는 호출을 만들 수 있다. 클래스 객체가 CallStaticObjectMethod 함수에

전달한다는 것을 기억하라.

```
jobject obj_ref = (*env)->CallStaticObjectMethod(env,class_System, id_getProperty,
        (*env)->NewStringUTF(env,"java.class.path") );
```

이 메소드의 반환 값은 jobject 타입이다. 만일 우리가 이것을 문자열로 조작하길 원한다면 jstring 으로 형변환을 해야 한다:

```
jstring str_rer = (jstring)obj_ret;
```

C++노트 : C에서 jstring, jclass 타입뿐만 아니라 뒤에서 소개할 배열 타입들은 모두 jobject 와 동일한 타입이다. 그러므로 앞선 예제에서 형변환은 C 에서 꼭 필요한 것은 아니다. 그러나 C++에서 이들 타입들은 올바른 상속 계층도를 갖는 "더미 클래스"에 대한 포인터로 정의된다. 예를 들면, jobject 에 jstring 을 할당하면 C++에서는 형변환없이 합법적이다. 그러나 jstring 에 jobject 를 할당하려면 형변환이 필요하다.

생성자

고유 메소드는 그 생성자를 호출하여 새로운 자바 객체를 생성할 수 있다. 이것은 다음 호출을 통해 이루어진다.

```
jobject obj_new = (*env) -> NewObject(env, class, methodID, construction parameters);
```

여러분은 GetMethodID 함수를 통해 이 호출에 필요한 메소드 ID 를 얻는다. 메소드 이름을 "<init>"으로 명시하고 생성자의 인코딩된 서명에 의해 GetMethodID 함수를 호출한다. 예를 들어, 다음은 고유 메소드가 FileOutputStream 객체를 생성하는 방법을 보여준다.

```
const char[] filename = " . . .";
jstring str_filename = (*env)->NewStringUTF(env, filename);
jclass class_FileOutputStream = (*env)->FindClass(env, "java/io/FileOutputStream");
jmethodID id_FileOutputStream =(*env)->GetMethodID(env,class_FileOutputStream,
        "<init>", "(Ljava/lang/String;)V");
jobject obj_stream = (*env)->NewObject(env,class_ FileOutputStream, id_FileOutputStream,
        str_fileName);
```

생성자의 서명은 java.lang.String 타입의 전달인자를 취하고 void 타입을 반환한다.

메소드 호출 대안

고유 코드에서 자바 메소드를 호출하기 위한 JNI 함수의 몇몇 변형이 존재한다. 이들은 이전에 논의했던 함수만큼 중요한 것은 아니지만 때때로 유용하다.

CallNonvirtualXxxMethod 함수들은 암시적인 전달인자와 명시적인 전달인자, 메소드 ID, 클래스 객체를 받는다.(클래스 객체는 암시적인 전달인자의 상위 클래스에 해당해야 한다.) 함수 호출은 일반 동적 디스패치 메카니즘을 건너뛰고 특정 클래스내의 메소드 버전을 호출한다.

모든 호출 함수들은 배열이나 va_list 내의 극명한 전달인자를 받는 "A"와 "V"로 시작하는 버전을 갖는다.(va_list는 stdarg.h C 헤더 파일내에 정의된다.)

예제 11-14 : Printf3Test.java

```
import java.io.*;

class Printf3Test
{  public static void main(String[] args)
   {  double price = 44.95;
      double tax = 7.75;
      double amountDue = price * (1 + tax / 100);
      PrintWriter out = new PrintWriter(System.out);
      Printf3.fprint(out,
         "Amount due = %8.2f\n", amountDue);
      out.flush();
   }
}
```

예제 11-15 : Printf3.java

```
import java.io.*;

class Printf3
{  public static native void fprint(PrintWriter out,
      String format, double x);
   static
   {  System.loadLibrary("Printf3");
   }
}
```

예제 11-16 : Printf3.c

```c
#include "Printf3.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

char* find_format(const char format[])
/**
 * @param format a string containing a printf format specifier
 * (such as "%8.2f"). Substrings "%%" are skipped.
 * @return a pointer to the format specifier (skipping the '%')
 * or NULL if there wasn't a unique format specifier
 */
{  char* p;
   char* q;

   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
   /* now check that % is unique */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating point format */
   return p;
}

JNIEXPORT void JNICALL Java_Printf3_fprint
   (JNIEnv* env, jclass cl, jobject out, jstring format,
   jdouble x)
{  const char* cformat;
   char* fmt;
   jstring str;
   jclass class_PrintWriter;
   jmethodID id_print;

   cformat = (*env)->GetStringUTFChars(env, format, NULL);
   fmt = find_format(cformat);
   if (fmt == NULL)
      str = format;
   else
   {  char* cstr;
      int width = atoi(fmt);
      if (width == 0) width = DBL_DIG + 10;
      cstr = (char*)malloc(strlen(cformat) + width);
      sprintf(cstr, cformat, x);
      str = (*env)->NewStringUTF(env, cstr);
      free(cstr);
   }
   (*env)->ReleaseStringUTFChars(env, format, cformat);
```

```
   /* now call ps.print(str) */

   /* get the class */
   class_PrintWriter = (*env)->GetObjectClass(env, out);

   /* get the method ID */
   id_print = (*env)->GetMethodID(env, class_PrintWriter,
      "print", "(Ljava/lang/String;)V");

   /* call the method */
   (*env)->CallVoidMethod(env, out, id_print, str);
}
```

C 코드에서 자바 메소드 실행하기

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

  클래스내의 메소드의 식별자를 반환한다.

  전달인자:

env	JNI **인터페이스 포인터**
cl	**클래스 객체**
name	**메소드 이름**
sig	**인코딩된 메소드 서명**

- void CallXxxMethod(JNIEnv *env, jobject obj, jmethodID id, args)
- void CallXxxMethodA(JNIEnv *env, jobject obj, jmethodID id,jvalue args[])
- void CallXxxMethodV(JNIEnv *env, jobject obj, jmethodID id,va_list args)

  이들 함수들은 메소드를 호출한다. 반환 타입 Xxx **는** Object, Boolean, Byte, Char, Short, Int, Long, Float **또는** Double **중 하나이다. 첫번째 함수는 전달인자의 변수 수를 갖는다-메소드** ID **다음에 메소드 전달인자를 추가한다. 두번째 함수는** jvalue **배열내에 메소드 전달인자를 받는다. 여기서** jvalue **는 다음과 같이 정의된 유니온이다.**

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

} jvalue;

세번째 함수는 C 헤더 파일 stdargs.h 에 정의된 va_list 내의 메소드 전달인자를 받는다.

전달인자:

env	JNI 인터페이스 포인터
obj	메소드의 암시적 전달인자
id	메소드 식별자
args	메소드 전달인자

- void CallNonvirtualXxxMethod(JNIEnv *env, jobject obj,jclass cl, jmethodID id, args)
- void CallNonvirtualXxxMethodA(JNIEnv *env, jobject obj,jclass cl, jmethodID id, jvalue args[])
- void CallNonvirtualXxxMethodV(JNIEnv *env, jobject obj,jclass cl, jmethodID id, va_list args)

  이들 메소드는 동적 디스패치를 건너뛰고 메소드를 호출한다. 반환 타입 Xxx 는 Object, Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다. 첫번째 함수는 전달인자의 변수 수를 갖는다-메소드 ID 다음에 메소드 전달인자를 추가한다. 두번째 함수는 jvalue 배열내에 메소드 전달인자를 받는다. 세번째 함수는 C 헤더 파일 stdargs.h 에 정의된 va_list 내의 메소드 전달인자를 받는다.

  전달인자:

env	JNI 인터페이스 포인터
obj	메소드의 암시적 전달인자
cl	호출할 메소드의 구현을 갖는 클래스
id	메소드 식별자
args	메소드 전달인자

- jmethodID GetStaticMethodID(JNIEnv *env, jclass cl, const cha r name[], const char sig[])

  클래스내의 정적 메소드의 식별자를 반환한다.

  전달인자:

env	JNI 인터페이스 포인터
cl	클래스 객체
name	메소드 이름
sig	인코딩된 메소드 서명

- void CallStaticXxxMethod(JNIEnv *env, jobject obj, jmethodID id, args)
- void CallStaticXxxMethodA(JNIEnv *env, jobject obj, jmethodID id,jvalue args[])
- void CallStaticXxxMethodV(JNIEnv *env, jobject obj, jmethodID id,va_list args)

  이들 함수들은 정적 메소드를 호출한다. 반환 타입 Xxx 는 Object, Boolean, Byte,

Char, Short, Int, Long, Float 또는 Double 중 하나이다. 첫번째 함수는 전달인자의 변수 수를 갖는다-메소드 ID 다음에 메소드 전달인자를 추가한다. 두번째 함수는 jvalue 배열내에 메소드 전달인자를 받는다. 세번째 함수는 C 헤더 파일 stdargs.h 에 정의된 va_list 내의 메소드 전달인자를 받는다.

전달인자:	env	JNI 인터페이스 포인터
	cl	정적 메소드의 클래스
	id	메소드 식별자
	args	메소드 전달인자

- jobject NewObject(JNIEnv *env, jclass cl, jmethodID id, args)
- jobject NewObjectA(JNIEnv *env, jclass cl, jmethodID id, jvalue args[])
- jobject NewObjectV(JNIEnv *env, jclass cl, jmethodID id, va_list args)

  이 들 함수들은 생성자를 호출한다. 메소드 ID 는 "<init>" 메소드 이름과 void 반환 타입을 갖는 GetMethodID 에서 얻어진다.

  첫번째 함수는 전달인자의 변수 수를 갖는다-메소드 ID 다음에 메소드 전달인자를 추가한다. 두번째 함수는 jvalue 배열내에 메소드 전달인자를 받는다. 세번째 함수는 C 헤더 파일 stdargs.h 에 정의된 va_list 내의 메소드 전달인자를 받는다.

전달인자:	env	JNI 인터페이스 포인터
	cl	인스턴스화될 클래스
	id	생성자 메소드 식별자
	args	생성자 전달인자

배열

자바 프로그래밍 언어의 모든 배열 타입은 표 11-2 에서 보여지는 것처럼 해당하는 C 타입들을 가지고 있다.

표 11-2: 자바 배열 타입과 C 타입사이의 연결

자바 타입	C 타입
boolean[]	jbooleanArray
byte[]	jbyteArray
char[]	jcharArray
int[]	jintArray

short[]	jshortArray
long[]	jlongArray
float[]	jfloatArray
double[]	jdoubleArray
Object[]	jobjectArray

jarray 타입은 일반(Generic) 배열을 나타낸다.

C++노트 : C에서 이러한 모든 배열은 실제 jobject 의 동일 타입이다. 그러나 C++에서 이들은 그림 11-2 같은 상속 계층도내에 정렬된다.

그림 11-2 : 배열 타입들의 상속계층도

GetArrayLength 함수는 배열의 길이를 반환한다.

```
jarray array = ...;
jsize length = (*env)->GetArrayLength(env,array);
```

배열내의 항목들에 접근하는 방법은 배열이 객체들을 저장하는가 또는 기본 타입(bool,char, 또는 숫자 타입)을 저장하는 가에 달려있다. 여러분은 객체 배열내의 항목들을 GetObjectArrayElement 와 SetObjectArrayElement 메소드로 접근한다.

```
jobjectArray array = ...;
int i,j;
jobject x = (*env)->GetObjectArray(env,array,i);
(*env)->SetObjectArray(env,array,j,x);
```

이 접근방법은 단순한 반면에 분명 비효율적이다. 여러분은 직접 배열 항목들에 접근할 수 있도록 해주길 원한다. 특히, 벡터와 메트릭스 계산을 할 때 필요하다.

GetXxxArrayElements 함수는 배열의 시작 항목에 대한 C 포인터를 반환한다. 일반 문자열

에서 처럼, 여러분은 더 이상 해당 포인터가 필요하지 않을 때 ReleaseXxxArrayElements 함수를 호출하여 가상 머쉰에게 말해야 한다는 점을 기억해야 한다. 여기서 타입 Xxx 는 Object 가 아니라 기본 타입이어야 한다. 이러면 여러분은 직접 배열 항목을 읽고 쓸 수 있다. 그러나 포인터는 복사본을 가리키고 있기 때문에 여러분이 만든 변화들은 오직 해당하는 ReleaseXxxArrayElements 함수를 호출할 때 본래의 배열에 반영되었다는 것을 보장받을 수 있다.

노트 : 만일 GetXxxArrayElements 메소드의 세번째 전달인자로 jboolean 변수에 대한 포인터를 전달한다면 배열이 복사본인지 발견할 수 있다. 배열이 복사본이면 변수는 JNI_TRUE 로 채워진다. 만일 여러분이 해당 정보에 관심이 없다면 단지 NULL 포인터를 전달한다.

다음은 double 값의 배열에 있는 모든 항목들에 상수를 곱하는 예제 코드이다. 우리는 자바 배열에서 C 포인터 a 를 얻고 a[i]로서 개별적인 항목들에 접근한다.

```
jdoubleArray array_a = ...;
double scaleFactor = ...;
double* a = (*env)->GetDoubleArrayElements(env,array_a,NULL);
for( i = 0; i < (*env)->GetArrayLength(env,array_a); i++)
        a[i] = a[i] * scaleFactor;
(*env)-> ReleaseDoubleArrayElements(env, array_a, 0);
```

실제 가상 머쉰이 배열을 복사할 것인가는 어떻게 배열을 할당하고 가베지를 수집할 것인가에 달려있다. 몇몇 "복사" 가베지 수집자는 반복해서 객체들을 옮기고 객체 참조자를 갱신한다. 이러한 전략은 특정 위치에 배열을 고정하는 것에는 적합하지 않다. 왜냐하면 수집자는 고유 코드내의 포인터 값들을 갱신할 수 없기 때문이다. 그러나 JDK 에서 제공하는 현재 가상 머쉰상의 가베지 수집자는 복사 수집자가 아니다. 이 말의 의미는 현재 배열은 고정되어 있다는 것이다.

노트 : 선 JVM 구현에서, Boolean 배열은 꾸러미(packed) 배열로 표현된다. 그리고 GetBooleanArrayElements 메소드는 이들을 jboolean 값들의 비꾸러미(unpacked) 배열로 복사한다.

만일 여러분이 큰 배열에서 몇몇 항목들만 접근할 필요가 있다면 자바 배열로부터 일정한 범주의 항목들을 복사하고 작성하는 GetXxxArrayRegion 과 SetXxxArrayRegion 메소드를 사용한다.

여러분은 NewXxxArray 함수를 사용하여 고유 메소드들에 새로운 자바 배열을 생성할 수 있다. 객체들의 새로운 배열을 만들기 위해, 여러분은 길이와 배열 항목의 타입, 그리고 모든 항목들의 초기 항목(기본은 NULL 임)을 명시해야 한다. 다음은 예제이다.

```
jclass class_Employee = (*env)->FindClass(env, "Employee");
jobjectArray array_e = (*env)->NewObjectArray(env, 100, class_employee, NULL);
```

기본 타입들의 배열은 보다 단순하다. 여러분은 단지 배열의 길이를 제공한다.

```
JdoubleArray array_d = (*env)->NewDoubleArray(env, 100);
```

배열은 0 으로 채워진다.

C 코드에서 자바 배열 조작하기

- jsize GetArrayLength(JNIEnv *env, jarray array)
  배열내의 항목들의 수를 반환한다.
  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
- jobject GetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index)
  배열 항목들의 값을 반환한다.
  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
  index 배열 오프셋

- void SetObjectArrayElement(JNIEnv *env, jobjectArray array, jsize index, jobject value)
  배열 항목들을 새로운 값으로 설정한다.
  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
  index 배열 오프셋
  value 새로운 값

- Xxx* GetXxxArrayElements(JNIEnv *env, jarray array, jboolean* isCopy)
  자바 배열의 항목들에 대한 C 포인터를 만들어낸다. 필드 타입 Xxx 는 Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다. 포인터가 더 이상 필요하지 않을 때 ReleaseXxxArrayElements 에게 전달해야 한다.

  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
  isCopy 만일 복사가 되면 JNI_TRUE 로 채워진 jboolean 에 대한 포인터, 그렇지 않으면 JNI_FALSE 로 채워진다.

- void ReleaseXxxArrayElements(JNIEnv *env, jarray array, Xxx elems[], jint mode)
  GetXxxArrayElements 에 의해서 얻어진 포인터가 더 이상 필요하지 않다는 것을 가상 머쉰에게 알린다.
  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
  elems 더 이상 필요하지 않은 배열 항목들에 대한 포인터
  mode 0=배열 항목들을 갱신한 후에 elems 버퍼를 해제(Free)한다.
  JNI_COMMIT= 배열 항목들을 갱신한 후에 elems 버퍼를 해제하지 않는다.
  JNI_ABORT=배열 항목들을 갱신하지 않고 elems 버퍼를 해제한다.

- void GetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])

  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체
  start 시작 인덱스
  length 복사할 항목들의 수
  elems 항목들을 유지할 C 배열

- void SetXxxArrayRegion(JNIEnv *env, jarray array, jint start, jint length, Xxx elems[])
  C 배열에서 항목들을 자바 배열로 복사한다. 필드 타입 Xxx 는 Boolean, Byte, Char, Short, Int, Long, Float 또는 Double 중 하나이다.

  전달인자: env JNI 인터페이스에 대한 포인터
  array 배열 객체

start	시작 인덱스
length	복사할 항목들의 수
elems	항목들을 유지할 C 배열

오류 처리

고유 메소드는 자바 프로그램에서 중대한 보안 위험이다. C 런타임 시스템은 배열 경계 오류, 잘못된 포인터를 통한 인다이렉션 등에 대한 보호가 없다. 특히, 고유 메소드의 프로그래머는 자바 시스템의 무결성을 보전하기 위한 오류 조건을 모두 처리하여야 한다. 특히, 고유 메소드가 처리할 수 없는 문제를 진단할 때 반드시 그 문제를 자바 가상 기계에 알려야 한다. 자바 코드에서, 이러한 경우에 자연스럽게 예외(exception)를 던지게 될 것이다. 그러나, C 는 예외를 가지지 않는다. 대신에, 예외 객체를 생성하기 위해 Throw 나 ThrowNow 함수를 호출한다. 고유 메소드가 존재할 때, 자바는 그 예외를 던질 것이다.

Throw 함수를 사용하기 위해, Throwable 의 하위 타입의 객체를 생성하기 위해 NewObject 를 호출한다. 예를 들면, 다음은 EOFException 객체를 할당하고 이를 투구한다.

```
jclass class_EOFException = (*env)->FindClass(env,"java/io/EOFException");
jmethodID id_EOFException = (*env)->GetMethodID(env,class_EOFException,
                                        "<init>", "()V"); /* ID of default construct */
jthrowable obj_exc = (*env)->NewObject(env, class_EOFException, id_EOFException);
(*env)->Throw(env, obj_exc);
```

일반적으로 ThrowNew 를 호출하는 것이 보다 편리하다. ThrowNew 는 예외 객체와 주어진 클래스, UTF 문자열을 생성한다.

```
(*env)->ThrowNew(env, (*env)->FindClass(env,"java/io/EOFException"),
                        "Unexpected end of file");
```

Throw 와 ThrowNew 는 단지 예외상황을 전달한다. 이들 메소드는 고유 메소드의 제어 흐름을 중단시키지 않는다. 오직 메소드가 반환할 때 자바 가상 머쉰이 예외상황을 투구한다. 그러므로 Throw 와 ThrowNew 에 대한 모든 호출은 항상 return 구문을 수반해야 한다.

C++ 노트: 만일 C++로 고유 메소드를 구현한다면, 현재로서는 C++ 코드에서 자바 예외 객체를 던질 수 없다. C++ 바인딩에서, C++와 자바 예외 간의 변환을 구현하는 것이 가능하다. 그러나 불행히도, 이것은 현재 구현되어 있지 않다. 고유 C++메소드에서 자바 예외를 던지려면 Throw 와 ThrowNew 를 사용해야 하며 고유 메소드가 C++ 예외를 던지지 않도록 해야 한다.

일반적으로, 고유 코드는 자바 예외상황을 감지하는 데 관련하지 않는다. 그러나 고유 메소드가 자바 메소드를 호출할 때, 해당 메소드는 예외상황을 투구할 것이다. 더욱이 많은 JNI 함수들 또한 예외상화을 투구한다. 예를 들면, SetObjectArrayElement 는 인덱스가 바운드를 초과할 때 ArrayIndexOutOfBoundsException 을, 만약 저장된 객체의 클래스가 배열의 항목 클래스의 하위 클래스가 아니라면 ArrayStoreException 을 투구한다. 이러한 상황에서, 고유 메소드는 예외상황이 투구됐는지를 결정하기 위해 ExceptionOccured 메소드를 호출할 것이다. 다음 호출은 대기중인 예외상황이 없을 경우 NULL 을 반환하거나 현재의 예외상황 객체에 대한 참조를 반환한다.

```
jthrowable obj_exc = (*env)->ExceptionOccured(env);
```

만일 단지 여러분이 예외상황 객체에 대한 참조를 얻지않고 예외상황이 투구될지를 검사하고 싶다면, 다음을 사용하라.

```
jbool occurred = (*env)->ExceptionCheck(env);
```

일반적으로, 고유 메소드는 예외상황이 발생했을 때 가상머쉰이 자바 코드에게 이를 전파할 수 있도록 반환할 것이다. 그러나 고유 메소드는 예외상황을 처리할 수 있는지를 결정하기 위해 예외상황 객체를 분석할 수 있다. 만일 가능하다면 , 다음 함수는 예외상황을 해제하기 위해 호출돼야 한다.

```
(*env)->ExceptionClear(env);
```

다음 예제는 fprint 고유 메소드로, 고유 메소드 구현 시에 필요한 주의점을 가지고 구현되었다. 다음은 던질 예외이다.

- 만일 포맷 문자열이 NULL 이라면 NullPointerException
- 만일 포맷 문자열이 double 을 프린트할 때 적절한 % 지정자를 포함하지 않는 다면 IllegalArgumentException
- 만일 malloc 에 대한 호출이 실패한다면 OutOfMemeoryError

마지막으로, 고유 메소드로부터 자바 메소드를 호출할 때 예외상황을 검사하기 위한 방법을 보여주기 위해 우리는 스트림에 문자열을 한번에 하나씩 보내고 각 호출후에 ExceptionOccurred 를 호출한다. 예제 11-17 은 고유 메소드를 위한 코드를 보여준다. 그리고 예제 11-18 은 고유 메소드를 포함하고 있는 클래스의 정의를 포함하고 있다. 고유 메소드는 PrintWriter.print 에 대한 호출시 예외상황이 발생할 때 즉시 종료되지 않는다-먼저 cstr 버퍼를 해제한다. 고유 메소드가 반환할 때, 가상 머쉰은 반복해서 예외상황을 발생한다. 예제 11-19 의 테스트 프로그램은 어떻게 고유 메소드가 포맷팅 문자열이 유효하지 않을 때 예외상황을 투구하는 지를 보여준다.

예제 11-17 : Printf4.c

```
#include "Printf4.h"
#include <string.h>
#include <stdlib.h>
#include <float.h>

char* find_format(const char format[])
/**
 * @param format a string containing a printf format specifier
 * (such as "%8.2f"). Substrings "%%" are skipped.
 * @return a pointer to the format specifier (skipping the '%')
 * or NULL if there wasn't a unique format specifier
 */
{  char* p;
   char* q;

   p = strchr(format, '%');
   while (p != NULL && *(p + 1) == '%') /* skip %% */
      p = strchr(p + 2, '%');
   if (p == NULL) return NULL;
   /* now check that % is unique */
   p++;
   q = strchr(p, '%');
   while (q != NULL && *(q + 1) == '%') /* skip %% */
      q = strchr(q + 2, '%');
   if (q != NULL) return NULL; /* % not unique */
   q = p + strspn(p, " -0+#"); /* skip past flags */
   q += strspn(q, "0123456789"); /* skip past field width */
   if (*q == '.') { q++; q += strspn(q, "0123456789"); }
      /* skip past precision */
   if (strchr("eEfFgG", *q) == NULL) return NULL;
      /* not a floating point format */
   return p;
}

JNIEXPORT void JNICALL Java_Printf4_fprint
   (JNIEnv* env, jclass cl, jobject out, jstring format,
   jdouble x)
{  const char* cformat;
   char* fmt;
```

```
    jclass class_PrintWriter;
    jmethodID id_print;
    char* cstr;
    int width;
    int i;

    if (format == NULL)
    {  (*env)->ThrowNew(env,
         (*env)->FindClass(env,
         "java/lang/NullPointerException"),
         "Printf4.fprint: format is null");
       return;
    }

    cformat = (*env)->GetStringUTFChars(env, format, NULL);
    fmt = find_format(cformat);

    if (fmt == NULL)
    {  (*env)->ThrowNew(env,
         (*env)->FindClass(env,
         "java/lang/IllegalArgumentException"),
         "Printf4.fprint: format is invalid");
       return;
    }

    width = atoi(fmt);
    if (width == 0) width = DBL_DIG + 10;
    cstr = (char*)malloc(strlen(cformat) + width);

    if (cstr == NULL)
    {  (*env)->ThrowNew(env,
         (*env)->FindClass(env, "java/lang/OutOfMemoryError"),
         "Printf4.fprint: malloc failed");
       return;
    }

    sprintf(cstr, cformat, x);

    (*env)->ReleaseStringUTFChars(env, format, cformat);

    /* now call ps.print(str) */

    /* get the class */
    class_PrintWriter = (*env)->GetObjectClass(env, out);

    /* get the method ID */
    id_print = (*env)->GetMethodID(env, class_PrintWriter,
       "print", "(C)V");

    /* call the method */
    for (i = 0; cstr[i] != 0 && !(*env)->ExceptionOccurred(env);
            i++)
       (*env)->CallVoidMethod(env, out, id_print, cstr[i]);

    free(cstr);
}
```

예제 11-18 : Printf4.java

```
import java.io.*;

class Printf4
{  public static native void fprint(PrintWriter ps,
      String format, double x);
   static
   {  System.loadLibrary("Printf4");
   }
}
```

예제 11-19 : Printf4Test.java

```
import java.io.*;

class Printf4Test
{  public static void main(String[] args)
   {  double price = 44.95;
      double tax = 7.75;
      double amountDue = price * (1 + tax / 100);
      PrintWriter out = new PrintWriter(System.out);
      Printf4.fprint(out, "Amount due = %%8.2f\n", amountDue);
      out.flush();
   }
}
```

C 코드에서 오류 처리하기

- jint Throw(JNIEnv *env, jthrowable obj)

  고유 코드에서 빠져나올 때 투구할 예외상황을 준비한다. 성공이면 0 을 실패이면 음수를 반환한다.

  전달인자: env JNI 인터페이스 포인터
  obj 투구할 예외상황 객체

- jint ThrowNew(JNIEnv *env, jclass class, const char msg[])

  고유 코드에서 빠져나올 때 투구할 예외상황을 준비한다. 성공이면 0 을 실패이면 음수를 반환한다.

  전달인자: env JNI 인터페이스 포인터
  cl 투구할 예외상황 객체의 클래스
  msg 예외상황 객체의 String 생성 전달인자를 나타내는 UTF 문자열

- jthrowable ExceptionOccured(JNIEnv *env)

  예외상황이 지연되면 예외상황 객체를 반환하고 그렇지않으면 NULL 을 반환한다.

전달인자: env JNI 인터페이스 포인터

- jboolean ExceptionCheck(JNIEnv *env)
  예외상황이 지연되면 true 를 반환한다.
  전달인자: env JNI 인터페이스 포인터

- void ExceptionClear(JNIEnv *env)
  지연되는 모든 예외상황을 해제한다.
  전달인자: env JNI 인터페이스 포인터

호출 API

지금까지, C 가 보다 빠르고 자바 플랫폼에서 접근할 수 없는 기능에 접근하게 해주기 때문에 몇몇 C 호출을 하는 자바 프로그래임 언어로 작성한 프로그램을 고려했었다. 여러분이 반대의 상황에 있다고 가정해 보자. 여러분이 C 와 C++ 프로그램을 가지고 있고 자바 코드가 프로그램하기 쉽기 때문에 자바 코드에 대해 몇몇 호출을 하고 싶다. 물론, 여러분은 자바 메소드를 호출하는 방법을 알고 있다. 그러나 여전히 여러분은 자바 코드를 해석할 수 있도록 여러분의 프로그램에 자바 가상머쉰을 추가할 필요가 있다. 흔히 호출(Invocation) API 라고 불리는 것이 자바 가상 머쉰을 C 나 C++ 프로그램에 내장하기 위해 사용된다.

다음은 가상 머쉰을 초기화할 필요가 있는 최소화 코드이다.

```
JavaVMOption        options[1];
JavaVMInitArgs      vm_args;
JavaVM *jvm;
JNIEnv     *env;

options[0].optionString = "-Djava.class.path=.";

memset(&vm_args , 0, sizeof(vm_args));
vm_args.version = JNI_VERSION_1_2;
vm_args.nOptions =1;
vm_args.options = options;

JNI_CreateJavaVM(&jvm, (void**)&env, & vm_args);
```

JNI_CreateJavaVM 에 대한 호출은 가상머신을 생성하고 가상머쉰에 대한 jvm 포인터와 실행환경에 대한 env 포인터를 채운다.

여러분은 가상머쉰은 모든 옵션을 제공할 수 있다. 손쉽게 options 배열의 크기와 vm_args.nOptions 의 값을 증가시킨다. 예를 들면, 다음 옵션은 져스트-인-타임 컴파일러를 해제한다.

```
options[1].optionString = "-Djava.compiler=NONE";
```

팁: 여러분이 문제에 빠지거나 프로그램이 붕괴되거나, JVM 초기화가 거절되거나, 여러분의 클래스을 로드할 수 없을 때, JNI 디버거 모드를 구동시킨다. 다음 옵션을 설정한다.

```
options[1].optionString = "-verbose:jni";
```

여러분은 JVM 을 초기화하는 과정을 알려주는 수많은 메시지를 보게될 것이다. 만일 여러분이 로드된 여러분의 클래스를 보지 못한다면 여러분의 경로와 클래스 경로 설정을 검사하라.

일단 가상머쉰을 설정했다면 앞선 절에서 설명한 방법으로 자바 메소드를 호출할 수 있다: 손쉽게 일반적인 방법으로 env 포인터를 사용한다. 여러분은 오직 호출 API 내의 다른 함수 호출을 위해 jvm 포인터를 필요로 한다. 현재 이와 같은 함수는 오직 4 개 존재한다. 가장 중요한 것은 가상머쉰을 종료하는 함수이다:

```
(*jvm)->DestoryJavaVM(jvm);
```

예제 11-20 의 C 프로그램은 가상머쉰을 설정하고 1 권의 2 장에서 설명한 Welcome 클래스의 main 메소드를 호출한다.

주의 : 윈도우하에서, JVM 코드는 \jdk\jre\bin\classic\jvm.dll 이나 \jdk\jre\bin\hotpot\jvm.dll 이다. 여러분은 여러분의 경로에 이 디렉토리가 포함되도록 설정해야 한다. 만일 설정하지 않는다면, jvm.dll 파일이 없다고 불평할 것이다.

만일 여러분이 윈도우 구동 프로그램을 사용하여 자바 응용 프로그램을 전개한다면 여러분의 사용자가 jvm.dll 파일을 경로에 설정했다고 믿지않는 것이 좋다. 여러분은 여러

분의 사용자를 도울수 있고 직접 DLL 을 로드할 수 있다.(javac 와 java 프로그램이 이러한 작업을 수행한다.) 예제 코드를 위해, JDK 의 일부인 src.jar 파일내에 launcher\java_md.c 를 참조하라.

예제 11-20 : InvocationTest.c

```
#include <jni.h>
#include <stdlib.h>

int main()
{  JavaVMOption options[1];
   JavaVMInitArgs vm_args;
   JavaVM *jvm;
   JNIEnv *env;
   long status;

   jclass class_Welcome;
   jclass class_String;
   jobjectArray args;
   jmethodID id_main;

   options[0].optionString = "-Djava.class.path=.";

   memset(&vm_args, 0, sizeof(vm_args));
   vm_args.version = JNI_VERSION_1_2;
   vm_args.nOptions = 1;
   vm_args.options = options;

   status = JNI_CreateJavaVM(&jvm, (void**)&env, &vm_args);
   if (status == JNI_ERR)
   {  printf("Error creating VM\n");
       return 1;
   }

   class_Welcome = (*env)->FindClass(env, "Welcome");
   id_main = (*env)->GetStaticMethodID(env, class_Welcome,
     "main", "([Ljava/lang/String;)V");

   class_String = (*env)->FindClass(env, "java/lang/String");
   args = (*env)->NewObjectArray(env, 0, class_String, NULL);

   (*env)->CallStaticVoidMethod(env, class_Welcome,
     id_main, args);

   (*jvm)->DestroyJavaVM(jvm);

   return 0;
}
```

이 프로그램을 윈도우하에서 컴파일하기 위해 다음 명령어를 사용하라.

cl –Ic:\jdk\include –Ic: :\jdk\include\win32 InvocationTest.c
c:\jdk\lib\jvm.lib

여러분이 이 프로그램을 여러분의 고객에게 제공하길 원한다고 가정하자. 여러분은 다음의 아이템들을 제공해야 한다.

- 실행 프로그램(InvocationTest.exe)
- 자바 코드의 클래스 파일(Welcome.class)
- 자바 런타임 환경(JRE)

호출 API 함수

- jint JNI_CreateJavaVM(JavaVM** p_jvm, void** p_env, JavaVMInitArgs* vm_args)
  자바 가상머쉰을 초기화한다. 함수가 성공하면 0 을 반환하고 실패하면 JNI_ERROR 를 반환한다.
  전달인자: p_jvm 호출 API 함수 테이블에 대한 포인터로 채워진다.
  p_env JNI 함수 테이블에 대한 포인터로 채워진다.
  vm_args 가상머쉰 전달인자

- jint DestoryJavaVM(JavaVM** jvm)
  가상머쉰을 제거한다. 성공이면 0 을, 실패이면 음수를 반환한다. 이 함수는 가상머쉰 포인터를 통해 호출되애 한다. 예, (*jvm)-> DestoryJavaVM(Jjvm)
  전달인자: jvm 가상머쉰 포인터

완전한 자바 예제: 윈도우 레지스트리에 접근하기

이 절에서 우리는 이 장에서 논의한 모든 것을 포함하는 완전하게 작동하는 예제를 설명한다: 문자열과 배열, 객체들, 생성자 호출, 그리고 에러 처리를 하는 고유 메소드를 사용한다. 여러분에게 보여주는 것은 윈도우 레지스트리와 동작하는 C 기반의 API 의 일부셋을 자바 플랫폼에 둘러싸기 위한 방법이다. 물론 윈도우 레지스트리를 사용하는 프로그램은 본래 이식성이없다. 이러한 이유에서 표준 자바 라이브러리는 레지스트리를 지원하지 않는다. 그리고 이에 접근하기 위해 고유라이브러리를 사용하는 것은 당연하다.

윈도우 레지스트리 일반

윈도우 레지스트리에 익숙하지 않은 사람들을 위해: 윈도우 레이지스트리는 윈도우 95/98, NT 운영체제에 의해서 접근되는 데이터 저장소이다. 그리고 이것은 응용 프로그램을 위한 저장 영역으로서 이용할 수 있다.(좋은 참고 서적은 Ron Petrusha 의 inside the

Windows95 Registry, O' Reilly, 1996 이다.) 윈도우의 과거 버전에서, 운영 시스템뿐만 아니라 응용 프로그램은 흔히 INI 파일이라 불리는 것에 설정 전달인자를 저장한다. 윈도우 95,98 과 NT 프로그램은 대신 레지스트리를 사용하기로 되어 있다. 레지스트리는 많은 장점을 가지고 있다.

- INI 파일들은 문자열로 데이터를 저장한다. 레지스트리는 정수와 바이트 배열 같은 다른 데이터 타입들을 지원한다..
- INI 파일 섹션은 하부 섹션을 갖을 수 없다. 레지스트리는 완전한 트리 구조를 지원한다.
- 설정 전달인자들은 많은 INI 파일들상에 분산되어 있다. 이들을 레지스트리에 저장하는 것은 관리와 백업을 위한 단일 위치를 제공한다.

또한 레지스트리는 실패의 단일 위치이다.-만일 여러분이 레지스트리를 엉망으로 만든다면 여러분의 컴퓨터는 오작동을 일으키거나 부트시 실패한다! 이 절에서 제공하는 예제ㅗ 안전하다. 그러나 만일 여러분이 레지스트리에 어떤 변경하고자 한다면 처리하기 전에 레지스트리를 백업하는 방법을 배워야한다. 윈도우 레지스트리를 백업하는 방법에 대한 정보는 Petrusha 의 책을 참고하라.(또한 노턴 유틸리티 같은 프로그램은 레지스트리 백업을 위한 사용자 편의 프로그램을 제공한다.)

레지스트리를 검사하는 기본 도구는 레지스트리 편집기이다. 숙련되지 않아서 발생하는 잠재적인 에러와 열성적인 사용자 때문에 레지스트리 편집기를 구동하기 위한 아이콘이 없다. 대신, 도스 쉘(또는 시작|수행 박스에서 개방)을 시작하고 regedit 을 입력하라. 그림 11-3 은 작동되는 레지스트리 편집기를 보여준다.

그림 11-3 : 레지스트리 편집기

왼편은 트리 구조로 정렬되어 있는 키를 보여준다. 각 키는 다음 처럼 HKEY 노드들중의 하나로 시작한다.

HKEY_CLASSES_ROOT
HKEY_CURRENT_USER
HKEY_LOCAL_MACHI NE
…

오른편은 특정 키와 관련되어 있는 이름/값 쌍을 보여준다. 가령, 다음 키는

HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info

두개의 이름/값 쌍을 갖고 있다, 즉,

DefCompany=”Your Organization”

DefName=”Your Name”

이 경우, 값들은 문자열이다. 값들은 또한 정수나 바이트 배열이 될 수 있다.

레지스트리에 접근하기 위한 자바 플랫폼 인터페이스

우리는 자바 코드에서 레지스트리에 접근하는 단순한 인터페이스를 구현할 것이다. 그런 다음, 고유 코드로 이 인터페이스를 구현할 것이다. 우리의 인터페이스는 오직 소수의 레지스트리 연산만이 가능하며 코드 크기가 최소화되도록 했으며 키들의 추가,삭제, 열거같이 중요한 기능들은 생략하였다.

우리가 제공하는 제한된 하위셋으로도 여러분은 다음의 작업을 할 수 있다.

- 특정 키에 저장되어 있는 모든 이름들을 열거한다.
- 특정 이름으로 저장되어 있는 값을 읽는다.
- 특정 이름으로 저장되어 있는 값을 설정한다.

다음은 레지스트리 키를 캡슐화하는 자바 플랫폼 클래스이다.

```
public class Win32RegKey
{   public Win32RegKey(int theRoot, String thePath){…}
    public Enumeration names(){…}
    public native Object getValue(String name);
    public native void setValue(String name, Object value);

    public static final int HKEY_CLASSES_ROOT = 0x80000000;
    public static final int HKEY_CURRENT_USER = 0x80000001;
    public static final int HKEY_LOCAL_MACHINE = 0x80000002;
       …
}
```

names 메소드는 해당 키로 저장된 유지하고 있는 모든 이름들의 열거 객체를 반환한다. 여러분은 이들을 친숙한 hasMoreElements/nextElement 메소드로 얻을 수 있다. getValue 메소드는 문자열 , Integer 객체, 또는 바이트 배열로 객체를 반환한다. setValue 메소드의 값

전달인자 또한 이들 세개의 타입들중 하나이다.

다음은 해당 키에 저장되어 있는 문자열을 나열하는 단순한 기능이다.

HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info

```
public static void main(String[] args)
{  Win32RegKey key = new Win32RegKey(
        Win32RegKey.HKEY_CURRENT_USER,
        "Software\\Microsoft\\MS Setup (ACME)\\User Info");

      Enumeration enum = key.names();

      while (enum.hasMoreElements())
      {  String name = (String)enum.nextElement();
         System.out.print(name + " = " + key.getValue(name));

      }
}
```

이 프로그램의 기본적인 결과는 다음과 같다.

DefCompany=Horstmann Software

DefName=Cay Horstmann

고유 메소드로서 레지스트리 접근 함수 구현하기

우리는 세가지 기능을 구현할 필요가 있다.

- 특정 키의 값을 얻는다.
- 특정 키의 값을 설정한다.
- 특정 키의 이름들을 반복한다.

운좋게도, 여러분은 필요한 모든 도구들을 가지고 있다. 가령, 자바 문자열과 배열 사이의 변환 같은. 그리고 여러분은 무엇인가 잘못된 경우 자바 예외상황을 투구하는 방법을 살펴보았다.

두개의 이슈 때문에 이들 고유 메소드가 앞선 예제들보다 보다 복잡하다.getValue 와 setValue 메소드는 Object 타입을 다룬다. Object 타입은 String, Integer 또는 byte[]중에 하나가 될 수 있다. 그리고 열거객체는 hasMoreElements/nextElement 메소드에 대한 성공적인 호출사이의 상태를 저장할 필요가 있다.

먼저, getValue 메소드를 살펴보자. 코드(예제 11-22 에 보여진다.)는 다음의 과정들을 따른다.

1. 레지스트리 키를 개방한다. 그들의 값들을 읽기위해 레지스트리 API 는 키들의 개방을 필요로 한다.
2. 해당 이름과 관련된 값의 타입과 크기를 조회한다.
3. 데이터를 버퍼로 읽는다.
4. 만일 타입이 REG_SZ(문자열)이면 값 데이터를 갖는 새로운 문자열을 생성하기위해 NewStingUTF 를 호출한다.
5. 만일 타입이 REG_DWORD(32 비트 정수형)이면, Integer 생성자를 호출한다.
6. 만일 타입이 REG_BINARY 이면 새로운 바이트 배열을 생성하기 위해 NewByteArray 를 호출하고 SetByteArrayRegion 을 사용하여 값 데이터를 바이트 배열로 복사한다.
7. 만일 타입이 이들 타입중 어느것도 아니거나 API 함수를 호출할 때 에러가 발생하면 예외상황을 투구하고 현재 시점까지 얻은 모든 자원을 조심스럽게 해제한다.
8. 키를 닫고 생성한 객체(String, Integer, 또는 byte[])를 반환한다.

여러분이 볼 수 있듯이, 이 예제는 다른 타입의 자바 객체들을 생성하는 방법을 보여주는 좋은 예제이다.

이 고유 메소드에서, 일반(Generic) 반환 타입을 만나는 것은 어렵지 않다. jstring, jobject 또는 jarray 참조는 jobject 로 반환된다. 그러나 setValue 메소드는 Object 에 대한 참조를 받는다. 그리고 이를 문자열이나 정수 또는 바이트 배열로 저장하기 위해 정확한 타입을 결정해야 한다. 우리는 이러한 결정을 value 객체의 클래스에 질의하고 java.lang.String, java.lang.integer,byte[]에 대한 클래스 참조를 찾고, IsAssignableFrom 함수로 이들을 비교하여 결정한다.

만일 class1 과 class2 가 두개의 클래스 참조이면 다음 호출은

```
(*env)->IsAssignableFrom(env,class1,class2)
```

class1 과 class2 가 같은 클래스이거나 class1 이 class2 의 하위 클래스이면 JNI_TRUE 를 반환한다. Class1 의 객체에 대한 참조는 class2 로 형변환될 수 있다. 가령, 다음이

```
(*env)->IsAssignableFrom(env,
        (*env)->GetObjectClass(env,value),
        (*env)->FindClass(env, "[B" ])
```

true 이면 우리는 value 가 바이트 배열이라는 것을 알게된다.

다음은 setValue 메소드의 코드에 대한 개괄설명이다.

1. 작성을 위해 레지스트리 키를 개방한다.
2. 작성할 값의 타입을 찾는다.
3. 만일 타입이 String 이면, 문자들에 대한 포인터를 얻기위해 GetStringUTFChars 를 호출한다. 또한 문자열 길이를 얻는다.
4. 만일 타입이 Integer 이면, 포장 객체내에 저장되어 있는 정수를 얻기위해 intValue 메소드를 호출한다.
5. 만일 타입이 byte[]이면, 바이트드에 대한 포인터를 얻기위해 GetByteArrayElements 를 호출한다. 또한 문자열 길이를 얻는다.
6. 데이터와 길이를 레지스트리에 전달한다.
7. 키를 닫는다. 만일 타입이 String 이거나 byte[]이면 문자열이나 바이트에 대한 포인터를 해제한다.

마지막으로, 키들을 나열하는 고유 메소드들로 관심을 돌려보자. 이들은 Win21RegKeyNameEnumeration 클래스의 메소드들이다(예제 11-21 참조). 열거 과정이 시작될 때, 우리는 키를 개방해야 한다. 열거가 진행되는 동안, 키 핸들을 유지해야 한다. 즉, 키 핸들은 열거객체와 함께 저장되야 한다. 키 핸들은 DWORD 타입이고 32 비트이다. 따라서 자바 정수로 저장될 수 있다. 키 핸들은 열거 클래스의 hkey 필드내에 저장된다. 열거가 시작될 때, 필드는 SetIntField 로 초기화된다. 값을 읽기위해 GetIntField 를 연속해서 호출한다.

팁 : 이 예제가 보여주듯이, 자바 객체 필드를 사용하여 고유 상태 데이터를 저장하는 것은 고유 메소드를 구현하는 데 아주 유용하다.

이 예제에서, 우리는 열거 객체에 세개의 다른 데이터 아이템을 저장한다. 먼저 열거가 시작될 때, 우리는 이름/값 쌍의 숫자와 가장 긴 이름의 길이를 알기 위해 레지스트리에 질의를 할 수 있다. 이들 정보는 이름들을 유지하는 C 문자열 배열들을 할당하기 위해 필요하다. 이들 값들은 열거객체의 count 와 maxsize 필드에 저장된다. 마지막으로, -1 로 초기화되는 index 필드는 열거의 시작을 나타내고 다른 객체 필드는 0 으로 초기화된다. 각 열거 단계후 증가된다.

다음, 우리는 열거를 지원하는 고유 메소드들을 살펴보자. HasMoreElements 메소드는 단순하다.

1. index와 count 필드들을 추출한다.
2. 만일 인덱스가 –1 이면, startNameEnumeration 함수를 호출하고 키를 개방한후 숫자

와 최대 길이를 질의하고 hkey, count, maxsize, index 필드들을 초기화한다.

3. 만일 index가 count 보다 작으면 JNI_TRUE를 반환한다. 그렇지 않으면 JNI_FALSE를 반환한다.

NextElement 메소드는 다소 어려운 작업을 필요로 한다.

1. index와 count 필드들을 추출한다.
2. 만일 인덱스가 –1 이면, startNameEnumeration 함수를 호출하고 키를 개방한후 숫자와 최대 길이를 질의하고 hkey, count, maxsize, index 필드들을 초기화한다.
3. 레지스트리에서 다음 이름을 얻어온다.
4. index를 증가한다.
5. 만일 index가 count와 동일하면 키를 닫는다.

프로그램을 컴파일하기 위해, 여러분은 advapi32.lib 라이브러리를 링크해야 한다. 컴파일하기 전에, Win32RegKey와 Win32RegKeyNameEnumeration에 대해 javah를 수행해야 한다는 사실을 기억하라. 완전한 명령어 라인은 다음과 같다.

```
cl –Ic:\jdk\include –Ic: :\jdk\include\win32 –LD Win32RegKey.c advapi32.lib –FeWin32RegKey.dll
```

예제 11-23 은 우리의 새로운 레지스트리 함수들을 테스트하기 위한 프로그램을 보여준다. 우리는 키에 대한 세개의 이름/값 쌍, 문자열, 정수, 그리고 바이트 배열을 추가한다.

```
HKEY_CURRENT_USER\Software\Microsoft\MS Setup (ACME)\User Info
```

그런 다음 우리는 해당 키의 모든 이름들을 열거하고 그들의 값들을 추출한다. 프로그램의 출력 결과는 다음과 같다.

```
DefCompany=Horstmann Software
DefName=Cay Horstmann
Default user = Bozo the clown
Lucky number = 13
Small primes = 2 3 5 7 11 13
```

비록 이러한 이름/값 쌍을 해당 키에 추가하는 것은 아마 해를 주진 않을 것이다. 여러분은 이 프로그래을 수행한 다음에 이들을 제거하기 위해 레지스트리 편집기를 사용하길 원할 것이다.

예제 11-21 : Win32RegKey.java

```
import java.util.*;

public class Win32RegKey
{  public Win32RegKey(int theRoot, String thePath)
   {  root = theRoot;
      path = thePath;
   }
   public Enumeration names()
   {  return new Win32RegKeyNameEnumeration(root, path);
   }
   public native Object getValue(String name);
   public native void setValue(String name, Object value);

   public static final int HKEY_CLASSES_ROOT = 0x80000000;
   public static final int HKEY_CURRENT_USER = 0x80000001;
   public static final int HKEY_LOCAL_MACHINE = 0x80000002;
   public static final int HKEY_USERS = 0x80000003;
   public static final int HKEY_CURRENT_CONFIG = 0x80000005;
   public static final int HKEY_DYN_DATA = 0x80000006;

   private int root;
   private String path;

   static
   {  System.loadLibrary("Win32RegKey");
   }
}

class Win32RegKeyNameEnumeration implements Enumeration
{  Win32RegKeyNameEnumeration(int theRoot, String thePath)
   {  root = theRoot;
      path = thePath;
   }

   public native Object nextElement();
   public native boolean hasMoreElements();

   private int root;
   private String path;
   private int index = -1;
   private int hkey = 0;
   private int maxsize;
   private int count;
}

class Win32RegKeyException extends RuntimeException
{  public Win32RegKeyException() {}
   public Win32RegKeyException(String why)
   {  super(why);
   }
}
```

예제 11-22 : Win32RegKey.c

```c
#include "Win32RegKey.h"
#include "Win32RegKeyNameEnumeration.h"
#include <string.h>
#include <stdlib.h>
#include <windows.h>

JNIEXPORT jobject JNICALL Java_Win32RegKey_getValue
  (JNIEnv* env, jobject this_obj, jstring name)
{  const char* cname;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD type;
   DWORD size;
   jclass this_class;
   jfieldID id_root;
   jfieldID id_path;
   HKEY root;
   jobject ret;
   char* cret;

   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path",
      "Ljava/lang/String;");

   /* get the fields */
   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj,
      id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* open the registry key */
   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
      != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
         (*env)->FindClass(env, "Win32RegKeyException"),
         "Open key failed");
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return NULL;
   }

   (*env)->ReleaseStringUTFChars(env, path, cpath);
   cname = (*env)->GetStringUTFChars(env, name, NULL);

   /* find the type and size of the value */
   if (RegQueryValueEx(hkey, cname, NULL, &type, NULL, &size) != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
         (*env)->FindClass(env, "Win32RegKeyException"),
         "Query value key failed");
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return NULL;
   }
```

```
   /* get memory to hold the value */
   cret = (char*)malloc(size);

   /* read the value */
   if (RegQueryValueEx(hkey, cname, NULL, &type, cret, &size) != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Query value key failed");
      free(cret);
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return NULL;
   }

   /* depending on the type, store the value in a string,
      integer or byte array */
   if (type == REG_SZ)
   {  ret = (*env)->NewStringUTF(env, cret);
   }
   else if (type == REG_DWORD)
   {  jclass class_Integer = (*env)->FindClass(env,
        "java/lang/Integer");
      /* get the method ID of the constructor */
      jmethodID id_Integer = (*env)->GetMethodID(env,
        class_Integer, "<init>", "(I)V");
      int value = *(int*)cret;
      /* invoke the constructor */
      ret = (*env)->NewObject(env, class_Integer, id_Integer,
        value);
   }
   else if (type == REG_BINARY)
   {  ret = (*env)->NewByteArray(env, size);
      (*env)->SetByteArrayRegion(env, (jarray)ret, 0, size,
        cret);
   }
   else
   {  (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Unsupported value type");
      ret = NULL;
   }

   free(cret);
   RegCloseKey(hkey);
   (*env)->ReleaseStringUTFChars(env, name, cname);

   return ret;
}

JNIEXPORT void JNICALL Java_Win32RegKey_setValue
  (JNIEnv* env, jobject this_obj, jstring name, jobject value)
{  const char* cname;
   jstring path;
   const char* cpath;
   HKEY hkey;
   DWORD type;
   DWORD size;
   jclass this_class;
```

```
jclass class_value;
jclass class_Integer;
jfieldID id_root;
jfieldID id_path;
HKEY root;
const char* cvalue;
int ivalue;

/* get the class */
this_class = (*env)->GetObjectClass(env, this_obj);

/* get the field IDs */
id_root = (*env)->GetFieldID(env, this_class, "root", "I");
id_path = (*env)->GetFieldID(env, this_class, "path",
   "Ljava/lang/String;");

/* get the fields */
root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
path = (jstring)(*env)->GetObjectField(env, this_obj,
   id_path);
cpath = (*env)->GetStringUTFChars(env, path, NULL);

/* open the registry key */
if (RegOpenKeyEx(root, cpath, 0, KEY_WRITE, &hkey)
   != ERROR_SUCCESS)
{  (*env)->ThrowNew(env,
     (*env)->FindClass(env, "Win32RegKeyException"),
     "Open key failed");
   (*env)->ReleaseStringUTFChars(env, path, cpath);
   return;
}

(*env)->ReleaseStringUTFChars(env, path, cpath);
cname = (*env)->GetStringUTFChars(env, name, NULL);

class_value = (*env)->GetObjectClass(env, value);
class_Integer = (*env)->FindClass(env, "java/lang/Integer");
/* determine the type of the value object */
if ((*env)->IsAssignableFrom(env, class_value,
   (*env)->FindClass(env, "java/lang/String")))
{  /* it is a string--get a pointer to the characters */
   cvalue = (*env)->GetStringUTFChars(env, (jstring)value,
     NULL);
   type = REG_SZ;
   size = (*env)->GetStringLength(env, (jstring)value) + 1;
}
else if ((*env)->IsAssignableFrom(env, class_value,
   class_Integer))
{  /* it is an integer--call intValue to get the value */
   jmethodID id_intValue = (*env)->GetMethodID(env,
     class_Integer, "intValue", "()I");
   ivalue = (*env)->CallIntMethod(env, value, id_intValue);
   type = REG_DWORD;
   cvalue = (char*)&ivalue;
   size = 4;
}
else if ((*env)->IsAssignableFrom(env, class_value,
   (*env)->FindClass(env, "[B")))
```

```
   {  /* it is a byte array--get a pointer to the bytes */
      type = REG_BINARY;
      cvalue = (char*)(*env)->GetByteArrayElements(env,
         (jarray)value, NULL);
      size = (*env)->GetArrayLength(env, (jarray)value);
   }
   else
   {  /* we don't know how to handle this type */
      (*env)->ThrowNew(env,
         (*env)->FindClass(env, "Win32RegKeyException"),
         "Unsupported value type");
      RegCloseKey(hkey);
      (*env)->ReleaseStringUTFChars(env, name, cname);
      return;
   }

   /* set the value */
   if (RegSetValueEx(hkey, cname, 0, type, cvalue, size)
      != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
         (*env)->FindClass(env, "Win32RegKeyException"),
         "Query value key failed");
   }

   RegCloseKey(hkey);
   (*env)->ReleaseStringUTFChars(env, name, cname);

   /* if the value was a string or byte array, release the
      pointer */
   if (type == REG_SZ)
   {  (*env)->ReleaseStringUTFChars(env, (jstring)value,
         cvalue);
   }
   else if (type == REG_BINARY)
   {  (*env)->ReleaseByteArrayElements(env, (jarray)value,
         (byte*)cvalue, 0);
   }
}

static int startNameEnumeration(JNIEnv* env, jobject this_obj,
   jclass this_class)
/* helper function to start enumeration of names
*/
{  jfieldID id_index;
   jfieldID id_count;
   jfieldID id_root;
   jfieldID id_path;
   jfieldID id_hkey;
   jfieldID id_maxsize;

   HKEY root;
   jstring path;
   const char* cpath;
   HKEY hkey;
   int maxsize = 0;
   int count = 0;

   /* get the field IDs */
```

```
   id_root = (*env)->GetFieldID(env, this_class, "root", "I");
   id_path = (*env)->GetFieldID(env, this_class, "path",
      "Ljava/lang/String;");
   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize",
      "I");
   id_index = (*env)->GetFieldID(env, this_class, "index",
      "I");
   id_count = (*env)->GetFieldID(env, this_class, "count",
      "I");

   /* get the field values */
   root = (HKEY)(*env)->GetIntField(env, this_obj, id_root);
   path = (jstring)(*env)->GetObjectField(env, this_obj,
      id_path);
   cpath = (*env)->GetStringUTFChars(env, path, NULL);

   /* open the registry key */
   if (RegOpenKeyEx(root, cpath, 0, KEY_READ, &hkey)
      != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Open key failed");
      (*env)->ReleaseStringUTFChars(env, path, cpath);
      return -1;
   }
   (*env)->ReleaseStringUTFChars(env, path, cpath);

   /* query count and max length of names */
   if (RegQueryInfoKey(hkey, NULL, NULL, NULL, NULL,
      NULL, NULL, &count, &maxsize, NULL, NULL, NULL)
      != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Query info key failed");
      return -1;
   }

   /* set the field values */
   (*env)->SetIntField(env, this_obj, id_hkey, (DWORD)hkey);
   (*env)->SetIntField(env, this_obj, id_maxsize, maxsize + 1);
   (*env)->SetIntField(env, this_obj, id_index, 0);
   (*env)->SetIntField(env, this_obj, id_count, count);
   return count;
}

JNIEXPORT jboolean JNICALL
   Java_Win32RegKeyNameEnumeration_hasMoreElements
   (JNIEnv* env, jobject this_obj)
{  jclass this_class;
   jfieldID id_index;
   jfieldID id_count;
   int index;
   int count;
   /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
```

```
   id_index = (*env)->GetFieldID(env, this_class, "index",
      "I");
   id_count = (*env)->GetFieldID(env, this_class, "count",
      "I");

   index = (*env)->GetIntField(env, this_obj, id_index);
   if (index == -1) /* first time */
   {  count = startNameEnumeration(env, this_obj, this_class);
      index = 0;
   }
   else
      count = (*env)->GetIntField(env, this_obj, id_count);
   return index < count;
}

JNIEXPORT jobject JNICALL
   Java_Win32RegKeyNameEnumeration_nextElement
   (JNIEnv* env, jobject this_obj)
{  jclass this_class;
   jfieldID id_index;
   jfieldID id_hkey;
   jfieldID id_count;
   jfieldID id_maxsize;

   HKEY hkey;
   int index;
   int count;
   int maxsize;

   char* cret;
   jstring ret;

  /* get the class */
   this_class = (*env)->GetObjectClass(env, this_obj);

   /* get the field IDs */
   id_index = (*env)->GetFieldID(env, this_class, "index",
      "I");
   id_count = (*env)->GetFieldID(env, this_class, "count",
      "I");
   id_hkey = (*env)->GetFieldID(env, this_class, "hkey", "I");
   id_maxsize = (*env)->GetFieldID(env, this_class, "maxsize",
      "I");

   index = (*env)->GetIntField(env, this_obj, id_index);
   if (index == -1) /* first time */
   {  count = startNameEnumeration(env, this_obj, this_class);
      index = 0;
   }
   else
      count = (*env)->GetIntField(env, this_obj, id_count);

   if (index >= count) /* already at end */
   {  (*env)->ThrowNew(env,
         (*env)->FindClass(env,
         "java/util/NoSuchElementException"),
         "past end of enumeration");
      return NULL;
```

```
   }

   maxsize = (*env)->GetIntField(env, this_obj, id_maxsize);
   hkey = (HKEY)(*env)->GetIntField(env, this_obj, id_hkey);
   cret = (char*)malloc(maxsize);

   /* find the next name */
   if (RegEnumValue(hkey, index, cret, &maxsize, NULL, NULL,
      NULL, NULL) != ERROR_SUCCESS)
   {  (*env)->ThrowNew(env,
        (*env)->FindClass(env, "Win32RegKeyException"),
        "Enum value failed");
      free(cret);
      RegCloseKey(hkey);
      (*env)->SetIntField(env, this_obj, id_index, count);
      return NULL;
   }

   ret = (*env)->NewStringUTF(env, cret);
   free(cret);

   /* increment index */
   index++;
   (*env)->SetIntField(env, this_obj, id_index, index);

   if (index == count) /* at end */
   {  RegCloseKey(hkey);
   }

   return ret;
}
```

예제 11-23 : Win32RegKeyTest.java

```
import java.util.*;

public class Win32RegKeyTest
{  public static void main(String[] args)
   {  Win32RegKey key = new Win32RegKey(
        Win32RegKey.HKEY_CURRENT_USER,
        "Software\\Microsoft\\MS Setup (ACME)\\User Info");

      key.setValue("Default user", "Bozo the clown");
      key.setValue("Lucky number", new Integer(13));
      key.setValue("Small primes", new byte[]
        { 2, 3, 5, 7, 11 });

      Enumeration enum = key.names();

      while (enum.hasMoreElements())
      {  String name = (String)enum.nextElement();
         System.out.print(name + " = ");

         Object value = key.getValue(name);
```

```
        if (value instanceof byte[])
        {  byte[] bvalue = (byte[])value;
           for (int i = 0; i < bvalue.length; i++)
              System.out.print((bvalue[i] & 0xFF) + " ");
        }
        else System.out.print(value);

        System.out.println();
     }
  }
}
```

형 조회 함수들

- jboolean IsAssignableFrom(JNIEnv *env, jclass cl1, jclass cl2)

  만일 첫번째 클래스의 객체가 두번째 클래스의 객체에 할당될 수 있다면 JNI_TRUE를 반환하고 그렇지 않으면 JNI_FALSE를 반환한다. 클래스들이 같거나 cl1이 cl2의 하위 클래스이거나 cl2가 cl1에 의해서 구현된 인터페이스를 표현하고 있거나 상위클래스중의 하나인 경우에 해당한다.

  전달인자: env JNI 인터페이스 포인터
  cl1, cl2 클래스 참조

- jclass GetSuperClass(JNIEnv *env, jclass cl)

  해당 클래스의 상위 클래스를 반환한다. 만일 cl이 Object 클래스나 인터페이스를 표현한다면 NULL을 반환한다.

  전달인자: env JNI 인터페이스 포인터
  cl 클래스 참조