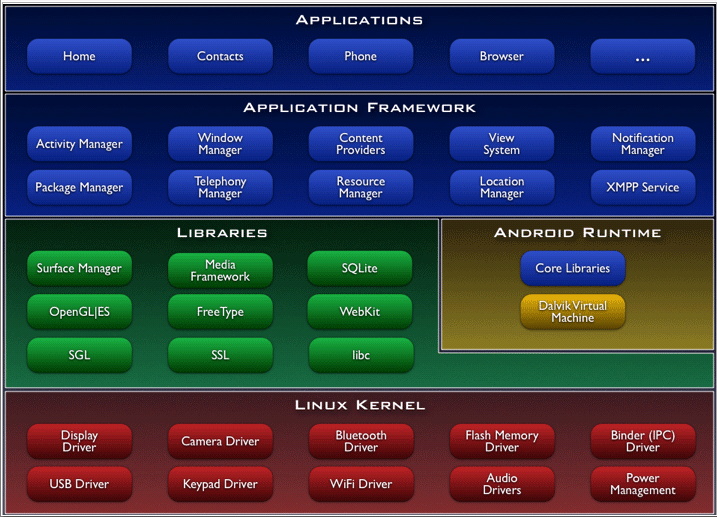
**GS1 한유선**

**4.1 안드로이드와 JNI**

**4.1.1 왜 안드로이드에서 JNI를 알아야 하는가?**



- JNI(Java Native Interface) : JNI는 자바 이외의 언어로 만들어진 모듈이나 어플리케이션과 자바 클래스가 서로 상호작용할 수 있게 정의한 인터페이스

- NDK(Native Development Kit) 안드로이드 어플리케이션에서 사용할 네이티브 라이브러리를 작성하기 위한 kit

- 사용하는 경우

a. 빠른 처리 속도를 요구하는 루틴 작성

b. 하드웨어 제어

c. 기존 C/C++ 프로그램 재사용

**4.2 JNI의 기본 원리 이해**

**4.2.1 자바에서 C 라이브러리 함수 호출하기**

a. 자바코드 작성

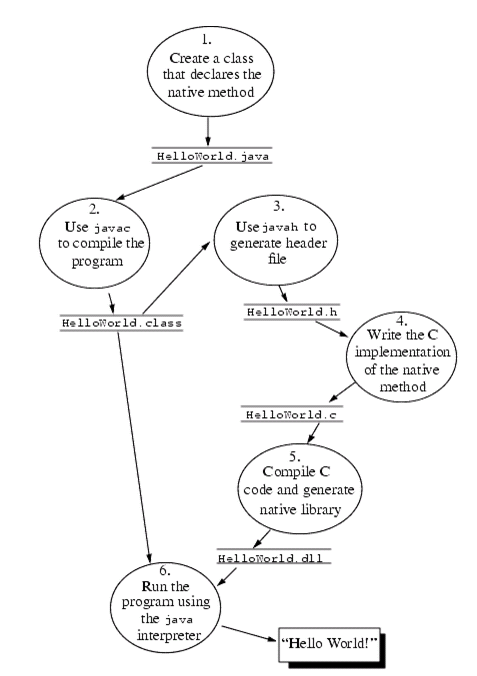
b. 자바 코드 컴파일

c. C 헤더 파일 생성

d. C 코드 작성

e. C 공유 라이브러리 생성

f. 자바 프로그램 실행



a. 자바 코드 작성

**public** **class** HelloJni **extends** Activity

{

/\*\* Called when the activity is first created. \*/

@Override

**public** **void** onCreate(Bundle savedInstanceState)

{

**super**.onCreate(savedInstanceState);

TextView tv = **new** TextView(**this**);

tv.setText( stringFromJNI() );

setContentView(tv);

}

**public** **native** String stringFromJNI();

**public** **native** String unimplementedStringFromJNI();

**static** {

System.*loadLibrary*("hello-jni");

}

}

<플랫폼별 System.loadLibrary() 메서드가 로딩하는 C 라이브러리 형식>

|  |  |  |
| --- | --- | --- |
| 호스트 플랫폼 | System.loadLibrary() 메서드에 전달한 인자 | 실제 로드되는 C 라이브러리 |
| 윈도우 | hello-jni | hellojni.dll |
| 리눅스 | hello-jni | libhellojni.so |

일반적으로 자바에서 네이티브 라이브러리를 로드하는 방법은 예제와 같이 static block을 사용하는 것이다. 메서드를 호출하는 시점보다 먼저 라이브러리를 로딩하기 위함이다.

b. 작성한 자바 코드 컴파일

- “javac HelloJni.java” 명령 입력 (자바 소스 컴파일) -> HelloJni.class 파일 생성됨

이 상태에서 자바 코드를 실행하면 로드할 라이브러리가 없기 때문에 에러가 난다.

c. C/C++ 헤더 파일 생성

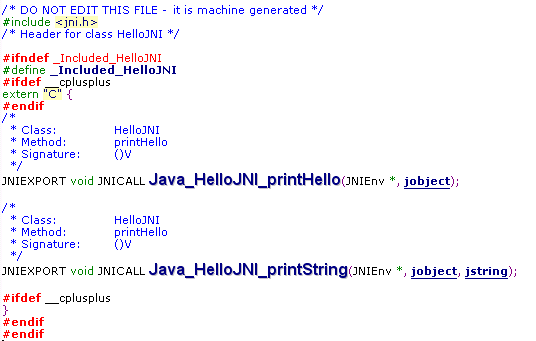
- 자바 클래스에서 선언한 이름대신 JNI 명세에 나온 대로 함수 원형을 만들어야 한다.

- 이러한 작업을 위해 javah 툴을 이용한다. 자바 네이티브 메서드와 연결될 수 있는 C 함수의 원형을 생성해 준다.

- 사용법: javah <native로 선언된 메서드를 포함한 자바 클래스 이름>

- “javah Hellojni” 명령 입력(javah 실행) -> HelloJni.h 파일 생성됨

<만들어진 헤더파일>



-JNIEXPORT, JNICALL : 이 매크로들은 플랫폼 종속적인 부분

<JNI 네이밍 룰>

- 디폴트 매개변수

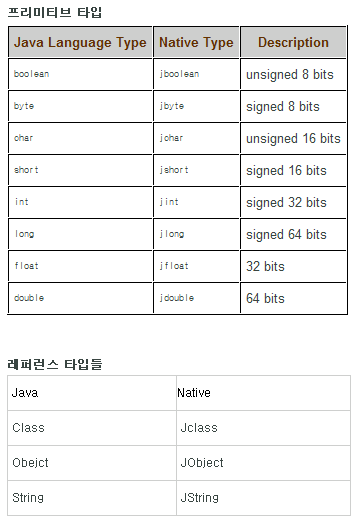
a. JNIEnv \* : JNI 인터페이스 포인터, 이 포인터를 통해 JNI 명세에 포함된 JNI 함수를 호출한다.

b. jobject : JNI에서 제공되는 *자바 네이티브 타입*으로, C 코드에서 자바 객체에 접근할 때 쓰인다. 여기에는 네이티브 메서드를 호출한 객체의 레퍼런스 값이 전달된다.

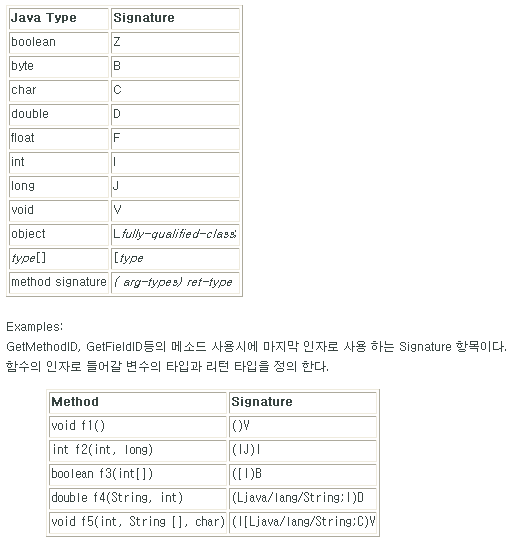
\*\* 자바 네이티브 타입 \*\*

자바는 플랫폼 독립적이여서 자바의 데이터 타입은 플랫폼 마다 메모리 크기가 동일하다. 그러나 네이티브 언어에서는 동일한 데이터 타입이더라도 플랫폼에 따라 크기가 달라질 수 있다.

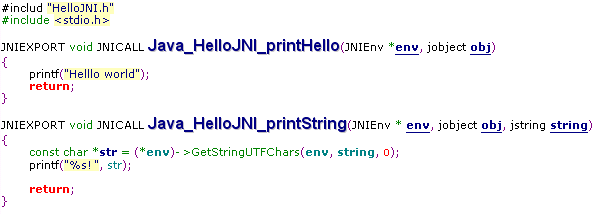
JNI는 네이티브 언어에서도 자바의 데이터 타입을 사용할 수 있게끔 자바 네이티브 타입을 제공해 준다.



\*\* Signature \*\*



d. C/C++ 코드 구현



- 헤더 파일의 함수 원형을 C파일에 복사하여 구현한다. 매개변수의 이름을 지정해 주어야 한다.

- printfHello() 메서드는 인자 없이 문자열 출력, printString() 메서드는 인자를 출력한다. 이 때 넘어온 문자열은 세 번째 매개변수인 jstring 타입의 변수에 저장된다.

- JNI 함수인 GetStringUTFChar()를 이용해 자바 String을 C언어의 문자열로 변환한다.

e. C 공유 라이브러리 생성

- 헤더파일과 소스 파일을 컴파일하여 공유 라이브러리를 생성한다.

f. 자바 프로그램 실행

**4.3 JNI 함수 이용하기**

JNI네이티브 함수에서 자바측 코드를 제어하는 방법

a. 자바 객체 생성

b. 클래스의 static member field에 접근하기

c. 클래스의 static method 호출하기

d. 자바 객체의 member field에 접근하기

e. 자바 객체의 method 호출하기

**4.3.1 자바측 코드 살펴보기 (JniFuncMain.java)**



a. JniFuncMain 클래스

- static 블록으로 라이브러리 로드한다.

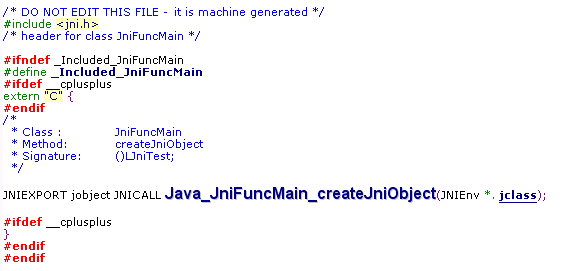
- 네이티브 메서드를 static으로 선언하여 객체 생성 없이 바로 사용한다.

b. JniTest 클래스

- 객체를 통해 호출될 메서드와 네이티브 코드에서 호출될 메서드를 구현하였다.

**4.3.2 JNI 네이티브 함수의 코드 살펴보기**

<헤더파일>



- 두번째 매개변수가 jobject가 아닌 jcalss인 이유: 메서드가 static으로 선언되어 있기 때문이다. 객체를 생성하지 않고도 클래스를 통해 바로 호출이 가능하기 때문에 두번째 매개변수가 객체가 아닌 클래스인 것이다.

<jnifunc.cpp 파일>



< JNI를 통한 멤버 필드 값 얻어오기 >

이 과정은

1 .접근하려는 멤버변숙가 선언된 자바 클래스의 jclass 값을 구하고,

2 .GetStaticFieldID(), GetFieldD() 메서드를 이용하여 jFieldID 값을 얻는다,

3. 1,2단계에서 구한 jclass와 jfieldID 값을 이용해 멤버 필드 값을 얻거나 설정한다.

- 예제에서는 createJniObject() 에 인자로 넘어온 clazz를 사용하면 돼지만 특정 클래스를 찾으려면 FindClass() 메서드를 이용하면 된다.

- GetStaticFieldId() 메서드는 함수 원형과는 다르게 첫 번째 env 인자를 넘기지 않고 있다. 그 이유는 JNI 함수를 호출하는 방식이 C/C++에 따라 다르기 때문이다. C 코드에서는 env 인자를 넘겨주지만, C++ 코드에서는 넘겨주지 않는다.

\*\* JNI에서 멤버 필드나 메서드의 시그너쳐 구하기

- 커맨드 창에 “javap –s –p JniFuncMain” 입력

-s 자바 시그너처를 출력,

-p 모든 클래스 및 멤버 출력

JniFuncMain 클래스 이름

< 객체 생성하기 >

JNI 네이티브 함수에서 자바 객체를 생성하는 지를 보여주는 코드이다.

이 과정은

1. 자바 객체를 생성하는 데 필요한 자바 클래스의 jclass 값을 구하고,

2. 자바 클래스 생성자의 메서드 ID 값을 구한 다음

3. 1,2에서 구한 jclass 값과 생성자 메서드로 자바 객체를 생성한다.

- 우선 FindClass()를 이용해서 객체 생성에 필요한 클래스를 로드한다. 인자는 객체생성을 할 클래스 이름이다.

- 생성자의 method ID를 구할 때는 GetMethodID()를 이용한다. 인자로는 자바 클래스와, 메서드 이름, 메서드 시그니처가 필요하다. 예제에서는 생성자를 찾기 때문에 메서드 이름에 <init>를 넣는다.

- newObject()를 이용해 객체를 생성한다. JniTest 클래스 생성자는 int형 인자를 받기 때문에 NewObject() 호출 시 100이라는 int형 값을 인자로 넘겼다.

< 자바 메서드 호출하기>

생성자를 호출하는 방식과 비슷하다.

- callByNative 는 리턴타입이 int 이기 때문에 CallIntMethod 함수로 메서드를 호출한다.

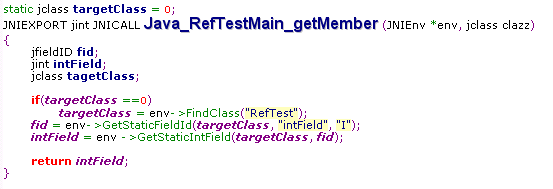
< JNI를 통한 멤버 필드 값 설정하기>

만들어진 객체의 멤버 필드 값을 변경시킨다. 방법은 멤버 필드 값을 읽어오는 것과 유사하다.

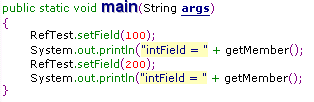
\*\* 지역 레퍼런스와 전역 레퍼런스

**잘못된 사용 예)**

<네이티브 코드에 구현되어 있는 getMember 함수>



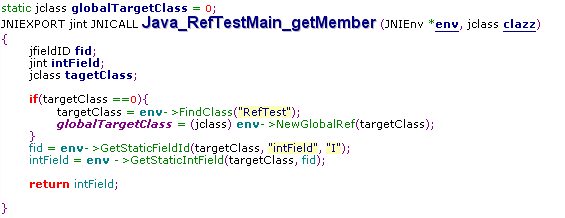
<getMemebr 함수를 호출해서 쓰는 부분>



JNI 네이티브 함수를 구현할 때 JNI 함수가 반환값으로 생성하는 jclass, jobject 등의 레퍼런스 값은 로컬 레퍼런스에 해당한다. 따라서 함수가 종료되고 나면 레퍼런스 값은 무효가 된다..

이런 이유로 JNI 네이티브 함수에서 레퍼런스 값을 전역적으로 사용할 수 있게 전역 레퍼런스를 제공한다. NewGlobalRef() 함수를 통해 생성할 수 있다.

**올바른 코드)**



DeleteGlobalRef() JNI 함수를 통해 레퍼런스 값을 소멸시킬 때까지 유효하다.

**4.4 C프로그램에서 자바 클래스 실행하기**

- 지금까지는 자바가 메인 프로그램이고 자바 쪽 코드에서 네이티브 메서드를 통해 C 함수를 호출해서 JNI를 이용하는 방식이었다. 여기서는 반대로 C/C++로 구현된 메인 어플리케이션에서 자바 클래스를 실행하는 JNI이용 방식을 알아본다.

*KeyPoint : 자바 클래스를 동작시키기 위해서 자바 가상 머신이 필요하다. 이를 위해 JNI는 네이티브 어플리케이션이 자신의 메모리 영역 내에 JVM을 로드할 수 있는 invocation API를 제공한다.*

- 사용하는 경우

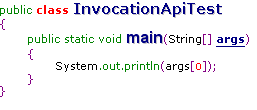
a. C/C++ 프로그램에서 기존에 작성한 자바 기반의 라이브러리를 이용하고 싶을 때

b. C/C++ 프로그램에서 자바 표준 라이브러리를 사용하고 싶을 때

c. 기존의 C/C++ 프로그램이 자바 프로그램과의 상호작용이 자주 필요로 할 때. 이 경우 invocation API를 이용해서 기존 프로그램과 자바 프로그램을 하나이ㅡ 프로그램으로 만들 수 있다.

- 달빅 가상 머신의 런처 프로그램도 invocation API를 통해 작성돼 있다.

**4.4.1 자바 코드**



.**4.4.2 C 코드**

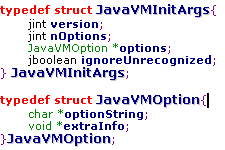


< 1. JVM에 전달할 옵션값을 생성하는 부분>

- JVM에 넘길 옵션 값을 생성할 때 필요한 구조체

a. JavaVMInitArgs

b. JavaVMOption



- JavaVMInitArgs 구조체는 내부적으로 JavaVMOption 구조체 배열을 포함한다. 즉 JavaVMOption 구조체가 각 옵션 값을 나타내고 JavaVMInitArgs가 이런 옵션들을 묶어 JVM에 전달한다.

- JavaVMInitArgs 구조체 필드

a. version: JVM에 넘길 옵션의 매개변수의 형식을 지정한다. jni.h 파ㅏ일에 정의되어 있다.

b. nOptions: JavaVMInitArgs에서 가리킬 JavaVMOption 구조체 배열의 원소 개수.

c. options: JavaVMInitArgs에서 가리킬 JavaVMOption 구조체 배열의 주소를 지정

d. ignoreUnrecognized: JVM이 잘못 정의된 옵션 값을 만났을 경우 무시하고 계속 진행할지 여부.

- JavaVMOption 구조체 필드

a. optionString: JVM의 옵션형식으로 문자열을 지정해 준다.

b. extraInfo: ?

<2. JVM 로드하는 부분>

- invocation API인 JNI\_CreateJavaVM()을 호출하여 JVM을 로드한다. (초기화도 동반)

*첫 번째 인자는 JVM을 생성하거나 종료하는 데 사용하는 JVM 인터페이스이다. 두 번째 인자로는 JNI인터페이스 포인터의 주소가 저장된다. 두 번째 인자를 통해 JNI함수를 이용할 수 있다. 세 번째 인자는 JVM에 넘길 매개변수이다.*

<3. 클래스를 로드하는 부분>

<4.로드한 클래스의 main()메서들를 호출하기 위해 메서드 ID를 구하는 부분>

<5. main()메서드에 넘길 인자를 생성하는 부분>

- main()메서드는 public static void main(String[] args)의 형태이므로 String[] 타입을 인자로 사용한다. 예제 코드에서는 “Hello Invocation API!!”라는 C문자열을 main()으로 전달할 것이다. 따라서 NewStringUTF() JNI함수를 이용해 C문자열을 String 객체로 변환한다. 그리고 나서 NewObjectArray()메서드를 이용해 String 객체의 배열을 만들고 변환한 문자열로 초기화 한다

<6. 메인 메서드를 호출하는 부분>

<7. JVM을 종료하는 부분

**4.5 JNI 네이티브 함수 직접 등록하기**

- JVM은 네이티브 라이브러리의 함수 심볼을 검색해서 자바에서 선언된 네이티브 메서드와 시그너처가 일치하는 것을 찾은 다음 네이티브 메서드(자바 선언부분)와 실제 구현부분인 JNI네이티브 함수를 매핑한다.

- 개발자가 직접 등록하면 이 매핑 과정을 생략할 수 있어 로딩 속도를 향상시킨다. 또한 JVM에 식별할 수 있게끔 JNI 네이티브 함수의 이름을 JNI 네이밍 룰에 맞출 필요도 없어진다.

**4.5.1 네이티브 라이브러리 로드 시에 JNI 네이티브 함수 등록하기**

- System.loadLibrary() 메서드의 동작 방식 : JVM은 라이브러리 내에 JNI\_OnLoad()라는 함수가 ㄱ구현되어 있는지를 확인하고 포함되어 있으면 JNI\_OnLoad() 함수를 호출한다. 그렇지 않다면 함수의 심볼을 비교해 매핑 작업을 수행한다.

따라서 개발자가 네이티브 메서드와 JNI 네이티브 함수를 매핑하려면 JNI\_OnLoad()함수 안에서 RegisterNatives() 함수를 이용해 매핑을 처리해야 한다.

\*\*예제 소스는 다음 페이지

<1. 매핑할 JNI 네이티브 함수의 원형을 선언>

- 네이밍 룰을 맞출 필요가 없다. 단, JNI 네이티브 함수에서 상용하는 두 개의 공통 매개변수(JNIEnv, jobject)는 기존 방식과 동일하게 지정해 준다.

<2. JNI\_OnLoad() 구현 - JNI버전 확인>

- GetEnv()로 JNI버전을 확인한다. 해당 버전을 지원하지 않으면 에러를 반환하고, 라이브러리 로딩을 종료한다.

- GetEnv()는 두번째 인자인 env에 JNI인터페이스 포인터를 반환한다. 이 포인터를 이용하여 JNI함수를 호출한다.

<3. JNI\_OnLoad() 구현 – 네이티브 함수 매핑>

- 네이티브 함수는 RegisterNatives()메서드를 이용한다.

원형: jarray RegisterNatives(JNIEnv \*env, jclass clazz, const JNINativeMethod \*methods, jnit nMethods)

- 각 필드 설명

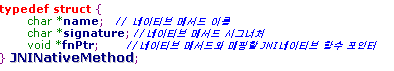
a. env: JNI인터페이스 포인터

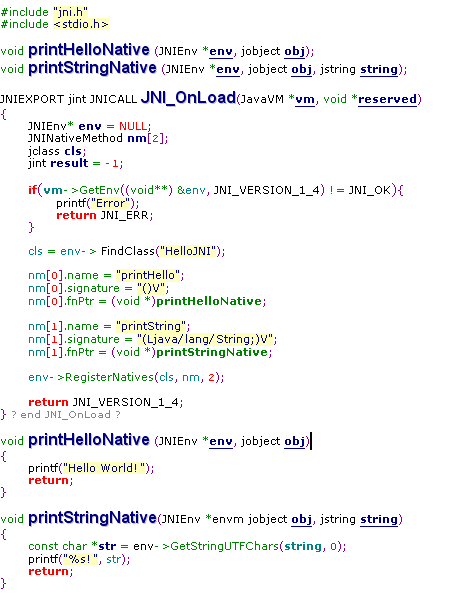
b. clazz: 자바 클래스

c. methods: 네이티브 메서드를 실제 JNI네이티브 함수와 연결시키기 위한 정보

d. nMethods: methods의 배열의 원소 개수

- 세번째 인자인 methods는 JNINativeMethod 타입의 배열로, 예제에서는 nm이라는 이름을 가지고 있다. JNINativeMethod 구조체는 다음과 같은 필드를 포함한다.





**4.5.2 안드로이드에서의 활용 예**