Programowanie natywne w systemie Android

Marcin Olszewski

1. Wstęp i cel ćwiczenia

Pisząc aplikacje działające w środowisku systemów mobilnych mamy do dyspozycji takie języki jak Javę w systemie Android, Objective-C dla iOS oraz C# dla Windows Phone. Jednak oprócz języków dedykowanych dla konkretnego systemu mobilnego mamy możliwość włączania do aplikacji kodu natywnego, czyli plików wykonywalnych EXE, DLL/SO z kodem maszynowym tworzonym przez tradycyjne kompilatory języków, np. C lub C++. Aplikacja natywna jest skompilowana pod konkretną platformę sprzętowo-programową, a więc działa na niej bezpośrednio, bez pomocy innych programów, takich jak emulatory czy maszyny wirtualne. Już na pierwszy rzut oka daje to korzyści wydajnościowe oraz pozwala na jeszcze większy dostęp do zasobów sprzętowych urządzenia przez programistę. Z drugiej jednak strony, kod aplikacji natywnej jest wykonywany bezpośrednio przez procesor, czyli jego uruchomienie polega na wykonaniu instrukcji skoku w miejsce, gdzie został załadowany w pamięci. Odbiera nam zatem zaletę przenośności i niezależności od platformy sprzętowej, jaką daje nam np. Java.
Celem ćwiczenia jest zapoznanie się studentów z procesem wytwarzania, kompilacji i zasadami działania aplikacji natywnych w systemie mobilnym Android z wykorzystaniem NDK.

2. Native Development Kit

Pakiet NDK(ang. Native Development Kit) dostarczony przez twórców Androida należy traktować jak uzupełnienie pakietu SDK(ang.), który dostarcza programistom zestawu narzędzi pozwalających na włączanie do aplikacji kodu natywnego, np. C lub C++. Pozwala to na optymalizację działania aplikacji oraz ponowne użycie kodu napisanego w tych językach, np. w postaci bibliotek. NDK stanowi uzupełnienie SKD, dla tego zanim zaczniemy implementację aplikacji natywnych, należy posiadać zainstalowane i skonfigurowane środowisko SDK. Dużą zaletą NDK jest dobra dokumentacja, która pozwala zrozumieć motywacje do użycia oraz działanie aplikacji natywnych. Oprócz szerokiej dokumentacji, pakiet zawiera wiele przykładowych projektów.
W skład pakietu, który można pobrać ze strony *https://developer.android.com/tools/sdk/ndk/* wchodzą:

- pliki nagłówkowe
- pliki C, C++
- prekompilowane biblioteki
- narzędzia do kompilowania, wiązania, analizowania oraz debugowania kodu,
- dokumentacja
- przykładowe aplikacje

Jako, że kod natywny dostowany jest pod konkretny procesor, dlatego NDK musi obejmować różne wersje narzedzi oraz prekompilowanych bibliotek – stosownie do obsługiwanych przez dany procesor instrukcji. Obecnie możemy liczyć na interfejsy ABI(ang. Application Binary Interface) dla:

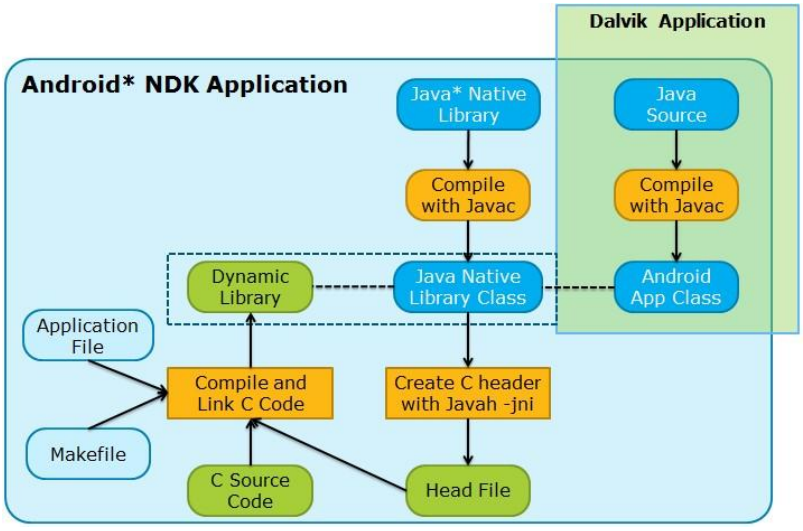
- arm64-v8a
- armeabi

- armeabi-v7a
- MIPS
- Mips64
- X86
- X86_64

3. Java Native Interface

Dedykowanym językiem dla systemu Android jest Java, dlatego wykorzystanie fragmentów kodu natywnego musi odbywać się dzięki mechanizmom adaptacji takiego kodu udostępnianym przez maszynę wirtualną Javy - JVM. Technologia umożliwiająca programowanie mieszane nazwa się JNI(ang. Java Native Interface). Dzięki JNI otrzymujemy wsparcie dla dwóch typów natywnego kodu – bibliotek natywnych oraz natywnych aplikacji. Możliwe jest to za pomocą natywnych metod z języka Java(słowo kluczowe *native*). Wywołując taką metodę programista nie zauważy różnicy w porównaniu do zwykłego wywołania metod. Dzięki JNI wykona się jednak kod zawarty w bibliotece natywnej – przygotowanej bezpośrednio dla procesora. Jednak trzeba pamiętać, że w ten sposób łączymy dwa różnie typowane języki(silnie typowaną Javę oraz słabo typowane C), zatem narażamy się także na błędy.
Zanim jednak wykorzystamy JNI, musimy pamiętać o tym, że:

- łączenie kodu Java oraz C nie odbywa się na poziomie leksykalnym, lecz dynamicznie na poziomie kodu wykonywalnego
- przechodząc na inną platformę sprzętową jesteśmy zmuszeni do rekompilacji części natywnej aplikacji
- błąd w kodzie natywnym bywa trudny do wykrycia
- warto skorzystać, gdy potrzebujemy dostępu zasobów sprzętowych niedostępnych w Javie
- warto skorzystać, kiedy chcemy napisać wrapper dla bibliotek natywnych, chociaż większość bibliotek natywnych ma już swoje odpowiedniki napisane w Javie
- dzięki JNI możemy zoptymalizować/wykonać nasz kod efektywniej



Two types of Android applications
https://software.intel.com/en-us/articles/android-application-development-and-optimization-on-the-intel-atom-platform

Powyższa ilustracja pomaga zrozumieć zależności pomiędzy modułami oraz proces kompilacji aplikacji natywnej. Oprócz standardowej aplikacji SDK dla Androida, aplikacja NDK wymaga plików nagłówkowych bibliotek .h, oraz ich kodów źródłowych .c. Kod plików nagłówkowych generujemy na podstawie klasy Javy dzięki opcji –jni programu javah. Na jego podstawie implementujemy źródło funkcji pamiętając o tym, że JNI wymusza użycie typów z jni.h. Poniższa tabela przedstawia odpowiednie typy zmiennych:

Java	Typ natywny	Rozmiar
boolean	jboolean	unsigned 8 bits
byte	jbyte	signed 8 bits
char	jchar	unsigned 16 bits
short	jshort	signed 16 bits
int	jint	signed 32 bits
long	jlong	signed 64 bits
float	jfloat	32 bits
double	jdouble	64 bits
void	void	

Następnie następuje proces kompilacji, do którego potrzebny jest plik make file – Android.mk i opcjonalnie Application.mk – który m.in. pozwala nam określić platformy na jakie będzie kompilowana wersja biblioteki. Tak skompilowana biblioteka dynamiczna musi zostać jeszcze załadowana z poziomu kodu Javy.

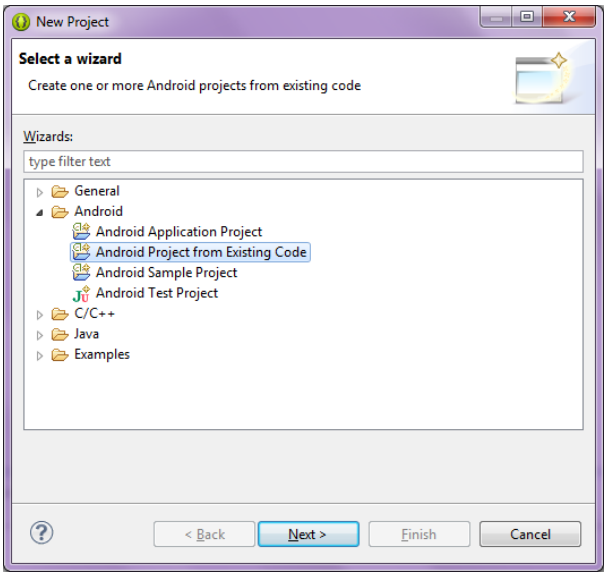
4. Zadania do samodzielnego wykonania

4.1. Kompilacja przykładowego projektu NDK

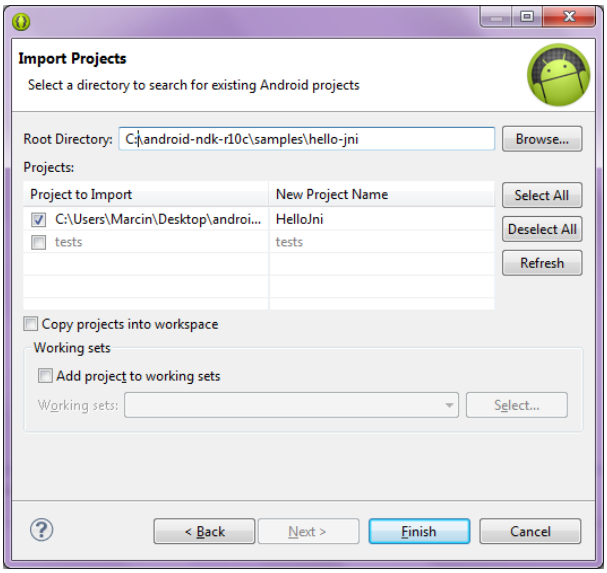
Aby zaprezentować proces wytwarzania aplikacji opartych o natywny kod, NDK zawiera szereg przykładów, które znajdują się w folderze **<ndk-path>\ samples**. Warto się z nimi zapoznać, ponieważ ilustrują one nie tylko, jak działają programy w NDK, ale także pokazują idee i dobre praktyki, które powinniśmy stosować, implementując własne biblioteki natywne. Są to zwykłe projekty w SDK, które implementują wykorzystanie bibliotek z języka C/C++. Sam projekt otwieramy w dowolnym środowisku programistycznym wspierającym SDK Androida, a kod źródłowy w C kompilujemy do biblioteki dynamicznej .so pod konkretną platformę na jakiej będziemy chcieli uruchomić nasz projekt. Rodzaj platformy musi być zgodny platformą emulatora lub urządzenia, w przeciwnym wypadku, nasza biblioteka nie zostanie w ogóle odnaleziona przez SDK.

Zadaniem studenta jest uruchomienie dowolnego przykładu katalogu **<ndk-path>\ samples**. Jako przykład zostanie wykorzystany projekt **<ndk-path>\ samples\hello-jni**, który jest prostą aplikacją pobierającą łańcuch znaków z metody natywnej i wyświetlającą go w interfejsie graficznym.

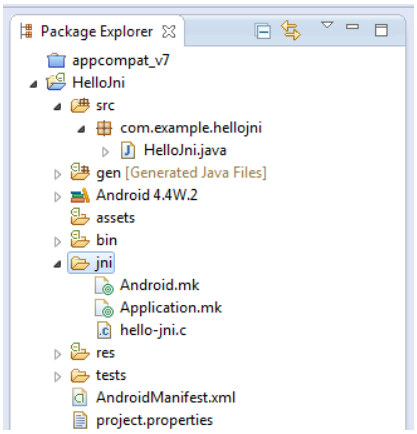
1. Po uruchomieniu IDE Ecplise, wybieramy File->New->Project
2. W następnym kroku wybieramy Adnroid->Android Project from Existing Code



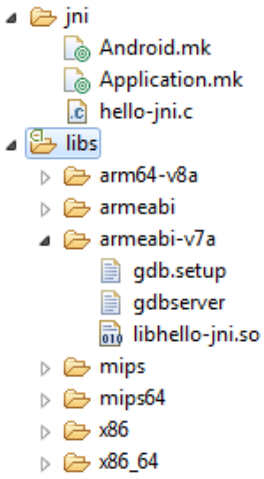
3. Wybieramy lokalizację projektu przykładowego, np. **<ndk-path>\ samples\hello-jni**



Pozostawiamy odznaczoną opcję importu projektu testowego, po czym wybierając Finish, powinniśmy w Package Explorer znaleźć nasz projekt. Folder /jni zawiera bibliotekę natywną – hello-jni.c, plik make file – Android.mk oraz opcjonalny plik, w którym określona jest wersja budowanej biblioteki – Application.mk.



Kolejnym etapem, jest kompilacja kodu źródłowego C do biblioteki dynamicznej, którą ładujemy w naszym projekcie SDK. W tym celu należy wykonać w głównym katalogu projektu polecenie **<ndk path>\ndk-build**. Po jego wykonaniu, oraz odświeżeniu drzewa projektów, pojawi się nowy folder **/libs**, w którym będą znajdowały się różne wersje naszej biblioteki, przeznaczone pod wszystkie obsługiwane w NDK platformy sprzętowe.



Dalej postępujemy już, jak w przypadku zwykłej aplikacji SDK, instalując projekt na wybranym urządzeniu fizycznym lub AVD.

4.2. Implementacja własnej biblioteki w języku C

Podejmując decyzję o wykorzystaniu bibliotek natywnych, należy jej świadomym. Nie powinniśmy używać programowania natywnego tylko dlatego, że jesteśmy dobrymi programistami języka C lub C++. Wykorzystanie bibliotek natywnych ma sens, gdy jesteśmy pewni, że biblioteka jest lepiej zaimplementowana niż jej odpowiednik w języku Java lub biblioteka natywna jest wydajniejsza niż biblioteki o podobnej funkcjonalności w Javie.

Po zapoznaniu się z przykładowymi projektami NDK, zadaniem studenta będzie zaimplementowanie własnej biblioteki języka C. Biblioteka powinna udostępniać funkcję wyliczającą n-tą wartość ciągu fibonachiego – rekurencyjnie lub iteracyjnie.

Po utworzeniu projektu SDK, należy dodać nową klasę, która może wyglądać następująco:

```
package pl.gda.pg.eti.geo.fibonacci;

public class FibonacciLib {

	public native static long fib(long n);

}
```

Słowo kluczowe `native` określa, że metoda zaimplementowana w języku zależnym od platformy, oraz używane jest tylko przy metodach. JVM wywołując taką metodę odwołuje się do mechanizmów pozwalających na komunikację np. z C lub C++.

Biblioteka będzie składała się z dwóch plików, nagłówkowego ***.h*** oraz pliku ***.c***.

Plik nagłówkowy biblioteki należy wygenerować przy pomocy programu Java, który znajduje się w katalogu JDK. Aby program prawidłowo wygenerował plik nagłówkowy trzeba podać

W tym celu tworzymy nowy katalog **/jni** (katalog nie może mieć innej nazwy, ponieważ polecenie ndk-build domyślnie kompiluje biblioteki z tego katalogu) w głównym folderze projektu i wywołujemy polecenie:

javah –jni –classpath bin/classes –d jni/ pl.gda.pg.eti.geo.fibonacci.FibonacciLib

Jeżeli wszytsko przebiegnie prawidłowo, w folderze **/jni** znajdzie się plik nagłówkowy o nazwie utworzonej z nazwy pakietowej klasy poprzez zamianę kropek na znaki podkreślenia, np.:

pl.gda.pg.eti.geo.fibonacci.FibonacciLib -> **pl_gda_pg_eti_geo_fibonacci_FibonacciLib.h**

Kolejnym krokiem jest utworzenie pliku .c na podstawie pliku nagłówkowego. W tym celu w katalogu **/jni** należy utworzyć nowy plik z rozszerzeniem .c o nazwie takiej samej, jak plik nagłówkowy.

Przykładowa zawartość pliku nagłówkowego może mieć postać:

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

#ifndef _Included_pl_gda_pg_eti_geo_fibonacci_FibonacciLib
#define _Included_pl_gda_pg_eti_geo_fibonacci_FibonacciLib
#ifdef __cplusplus
extern "C" {
#endif
/*
 * Class:     pl_gda_pg_eti_geo_fibonacci_FibonacciLib
 * Method:    fib
 * Signature: (J)J
 */

JNIEXPORT jlong JNICALL Java_pl_gda_pg_eti_geo_fibonacci_FibonacciLib_fib
  (JNIEnv *, jclass, jlong);

#ifdef __cplusplus
}
#endif
#endif
```

Zatem ciało pliku .c może wyglądać następująco:

```
#include <pl_gda_pg_eti_geo_fibonacci_FibonacciLib.h>

static jlong fib(jlong n){
       /*
        * Implement Here
        *
        * */
       }
```

```
JNIEXPORT jlong JNICALL Java_pl_gda_pg_eti_geo_fibonacci_FibonacciLib_fib
      (JNIEnv *env, jclass clazz, jlong n) {
            return fib(n);
}
```

Przed zbudowaniem biblioteki dynamicznej, należy utworzyć plik makefile o nazwie Android.mk z zawartością:

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE    := pl_gda_pg_eti_geo_fibonacci_FibonacciLib
LOCAL_SRC_FILES := pl_gda_pg_eti_geo_fibonacci_FibonacciLib.c

include $(BUILD_SHARED_LIBRARY)
```

oraz opcjonalnie plik Application.mk z zawartością:

```
APP_ABI := all
```

Po poprawnym skompilowaniu kodu biblioteki do bibliotek dynamicznych – analogicznie do poprzedniego zadania, należy załadować odpowiednią bibliotekę. W tym celu należy w bloku statycznym, np. w klasie zawierającej metodę natywną, załadować bibliotekę, podając jej nazwę.

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

Która wersja biblioteki zostanie użyta?
W tym momencie możemy już z poziomu SDK wywołać naszą metodę oraz zaprezentować jej rezultat w interfejsie graficznym.

***** Po zainstalowaniu aplikacji na emulatorze, możemy dzięki poleceniu **<sdk-path>/platform-tools/adb logcat** śledzić działanie emulatora.

4.3. Implementacja biblioteki - wrappera do biblioteki natywnej języka C

Kolejną motywacją do programowania natywnego urządzeń mobilnych jest wykorzystanie w projekcie biblioteki już istniejącej, biblioteki napisanej np. w języku C lub C++. Aby wykorzystać istniejącą bibliotekę natywną, nie modyfikujemy jej kodu na zgodny z JNI, lecz implementujemy klasę wrappera, która pośredniczy w wywołaniach funkcji bibliotek natywnych.

Zadaniem studenta będzie implementacja biblioteki, która będzie umożliwiała dostęp do funkcji istniejącej już biblioteki języka C - *libbmp* - z poziomu kodu w Javie. Zaimplementowana funkcja powinna korzystając z biblioteki libbmp utworzyć bitmapę o zadanych rozmiarach oraz zapisać ją w pamięci trwałej urządzenia.

W tym celu należy utworzyć nowy projekt analogicznie do poprzedniego zadania, po czym zaimplementować klasę udostępniającą natywną funkcję statyczną o prototypie:

```
public static native void createBmp(int width, int height, int depth);
```

Funkcja powinna utworzyć w pamięci trwałej urządzenia bitmapę o zadanych parametrach: wysokość, szerokość i głębokości kolorów(1,2,4,8,16 lub 32). Na podstawie utworzonej klasy, wygenerować plik nagłówkowy programem *javah* i umieścić go w folderze jni/libbmptest. Następnie należy utworzyć w tym samym folderze pliku .c o odpowiedniej nazwie, w którym znajdzie się implementacja wrappera do biblioteki *bmplib*, tworzącego bitmapę.

```
{
	bmpfile_t *bmp;

	bmp = bmp_create(width, height, depth);

	//utworzenie piksela: r,g,b,a
	rgb_pixel_t pixel = {128, 64, 0, 0};

	//instrukcje wyrysowujące piksele na bitmapie
	//wyrysowanie pojedynczego piksela:
	//bmp_set_pixel(bitmapa, x, y, piksel)

	//zapis bitmapy
	bmp_save(bmp, "correct_storage_path");
	//usunięcie bitmapy z pamięci
	bmp_destroy(bmp);

}
```

Drugim etapem jest skopiowanie plików z biblioteką (.h oraz .c) do folderu jni/libbmp.
Ostatnim etapem jest utworzenie plików make file dla Androida.
jni/libbmp/Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := libbmp
LOCAL_SRC_FILES := bmpfile.c
include $(BUILD_SHARED_LIBRARY)
```

jni/libbmptest/Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(CLEAR_VARS)
LOCAL_MODULE := PortingShared
LOCAL_C_INCLUDES := $(LOCAL_PATH)/../libbmp/
LOCAL_SRC_FILES := cookbook_chapter8_portingshared_NativeLib.c
LOCAL_SHARED_LIBRARIES := libbmp
LOCAL_LDLIBS := -llog
include $(BUILD_SHARED_LIBRARY)
```

jni/Android.mk:

```
LOCAL_PATH := $(call my-dir)
include $(call all-subdir-makefiles)
```

Przed uruchomieniem aplikacji należy załączyć w bloku statycznym w odpowiedniej kolejności dwie skompilowane biblioteki.

Uwaga: Przy każdej zmianie kodu biblioteki należy ją skompilować.
<ndk path>\ndk-build clean
<ndk path>\ndk-build

Aby skompilować biblioteki pod wszystkie platformy należy utworzyć dodatkowo plik Application.mk z odpowiednim wpisem, analogicznie, jak w poprzednim zadaniu.