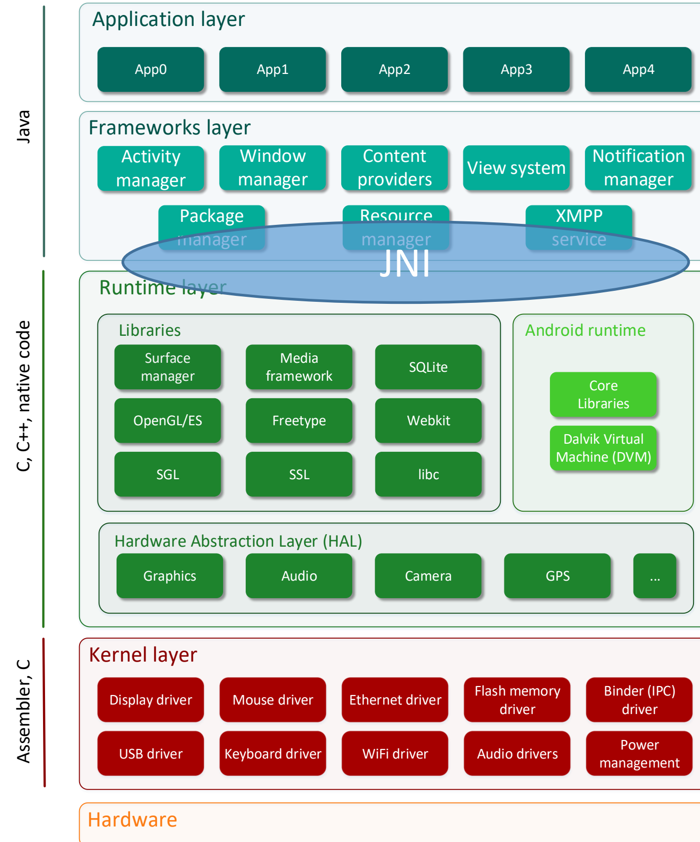
Projektovanje i arhitektura računarskih sistema

Java Native Interface - JNI

Odsek za računarsku tehniku i računarske komunikacije

Arhitektura Androida



Logička pozicija JNI u Android arhitekturi

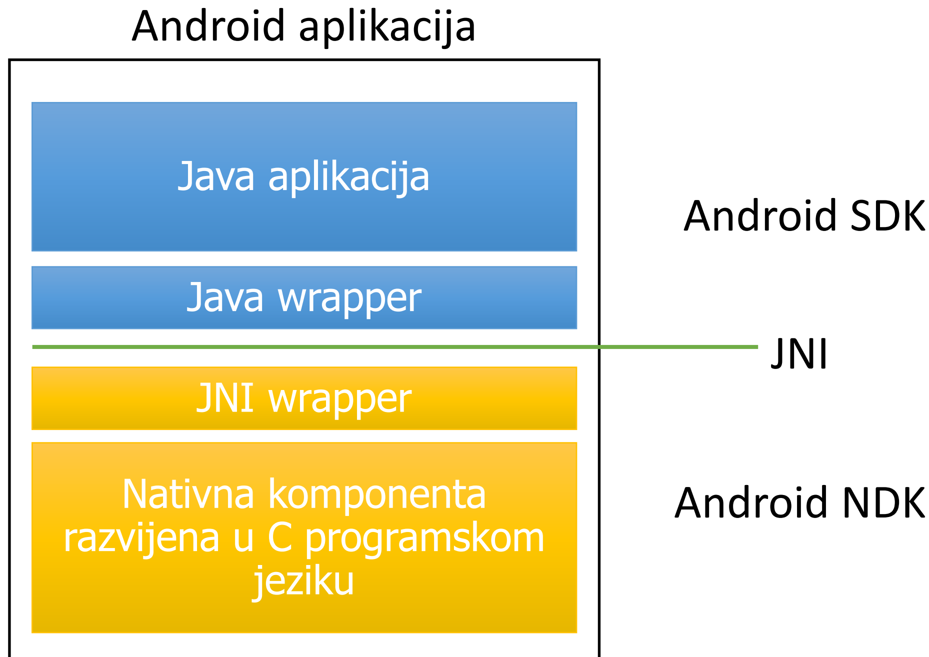
JNI

- Java Native Interface/Invocation
- Omogućuje da se iz Jave pozivaju C/C++ funkcije (nativne funkcije) koje i dalje čine deo aplikacije!
- Ceo Android radni okvir koristi ovaj koncept
  - Java kod je “plitak” i veoma brzo završi u nativnom kodu
  - Koristi se i u aplikacijama
- Nedostaci
  - greške koje mogu destabilizovati celu virtuelnu mašinu
  - gubi se platformska nezavisnost
  - ne radi automatski 'garbage collection’ objekata koji potiču iz nativnog koda, što može da dovede do programskih grešaka

Kada se koristi JNI?

- Performanse
  - Native kod je brži od Java koda
  - Pogodan za zahtevnu obradu
  - Intenzivna obrada podataka/soft real-time
- Zaštita
  - Java bytecode za značajno lakše vratiti u formu izvornog koda od nativnog
  - Zaštićene biblioteke
- Nešto što još nije implementirano u Javi
  - Pristup hardveru
  - Upotreba postojećih programa i biblioteka

Arhitektura softvera koji koristi JNI komponente



Mogućnosti nativnog dela programa

- Osim izvršavanja nativnih funkcija, iz nativnog koda se mogu realizovati i složene funkcionalnosti:
  - pristup atributima i metodama proizvoljne Java klase
    - čitanje i promena atributa nekog Java objekta
    - poziv metode nekog Java objekta
  - kreiranje Java objekata proizvoljne Java klase i poziv proizvoljnog konstruktora
  - pristup nizovima napravljenim u Javi
  - izazivanje izuzetaka (Exception) u Java kodu
  - itd.
- U suštini, može se uraditi sve što može i u Javi
  - Moderne Android igrice su često velikim delom razvijane u native modu, zbog prenosivosti na iOS, performansi, i ranijeg razvoja (postojećeg koda)

Primer u Java

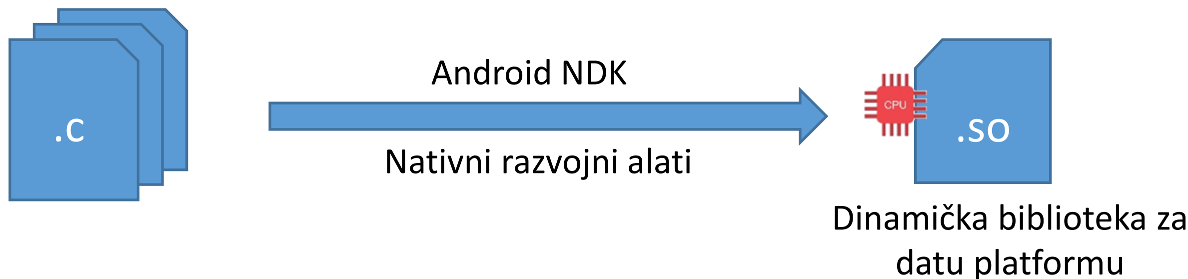
```
package com.example.myapplication;

import android.os.Bundle;
import android.widget.TextView;

public class MainActivity extends AppCompatActivity {
    // Used to load the 'native-lib' library on application startup.
    static {
        System.loadLibrary("native-lib");
    }
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        stringFromJNI(1);
    }

    /**
     * A native method that is implemented by the 'native-lib' native library,
     * which is packaged with this application.
     */
    public native String stringFromJNI(int test);
}
```

Izgleda kao integralni deo Java klase

Razvoj nativnih aplikacija/modula

- Ključna reč **native** u deklaraciji metode označava da je ta metoda nativna
- Sve nativne metode moraju da budu implementirane kao funkcije u dinamičkoj biblioteci (**.so** datoteke na Android platformu)
  - dinamičke biblioteke se učitavaju metodom System.loadLibrary("ime") u statičkom bloku aplikacije
- Da bi virtuelna mašina prepoznala nativnu funkciju u dinamičkoj biblioteci, potrebno je da ima odgovarajući potpis u prototipu
  - odgovarajući naziv, argumente i povratne vrednosti
  - potpis je obezbeđen mašinski generisanom .h datotekom, koja sadrži prototipove nativnih funkcija, sa odgovarajućim potpisom

Primer C implementacije nativnog modula




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

extern "C" JNIEXPORT jstring JNICALL
Java_com_example_myapplication_MainActivity_stringFromJNI__I
(JNIEnv* env, jobject jThis, jint i) {

    std::string hello = "Hello from C++";
    return env->NewStringUTF(hello.c_str());
}
```

Prototipovi nativnih funkcija

- Prototipovi nativnih funkcija imaju svoju prilično složenu šemu imenovanja
- Stoga je razvijen alat **javah** koji to olakšava
- Prototipovi nativnih funkcija generišu se pozivom **javah** alata na osnovu Java izvornog koda!
  - Na taj način se umanjuje mogućnost greške usled razlika između Java izvornog koda i prototipa
- Alat **javah** je standardni deo Java SDK
- Ovaj alat kao argument prima pun naziv klase (ime klase sa punim nazivom paketa); traži deklaracije nativnih metoda i generiše **.h** datoteku sa prototipovima svih nativnih funkcija
- Obratiti pažnju da nakon svake izmene Java datoteke sa nativnim metodama, OBAVEZNO pozvati alat **javah** i ponovo generisati C zaglavlje! Inače mogu nastati greške koje se veoma teško otkrivaju!

Prototipovi nativnih funkcija

- U Javi:

```
private native void f1(int i, float j, String s, double d, java.util.Vector v, SignatureDemo
     sd);
private native void f1(int[] i, float[] j, String[] s, double[] d, java.util.Vector[] v,
     SignatureDemo[] sd);
```

- U .h datoteci:

```
/* Method:    f1
 * Signature: (IFLjava/lang/String;DLjava/util/Vector;LSignatureDemo;)V
 */
JNIEXPORT void JNICALL
     Java_SignatureDemo_f1__IFLjava_lang_String_2DLjava_util_Vector_2LSignatureDemo_2
   (JNIEnv *, jobject, jint, jfloat, jstring, jdouble, jobject, jobject);
/* Method:    f1
 * Signature: ([I[F[Ljava/lang/String;[D[Ljava/util/Vector;[LSignatureDemo;)V
 */
JNIEXPORT void JNICALL
     Java_SignatureDemo_f1___3I_3F_3Ljava_lang_String_2_3D_3Ljava_util_Vector_2_3LSignatureDemo_
     2
   (JNIEnv *, jobject, jintArray, jfloatArray, jobjectArray, jdoubleArray, jobjectArray,
     jobjectArray);
```

Ova klasa nema odgovarajući tip u JNI – pogledati niže čime je predstavljena (jobject)

Java tipovi u JNI

- Za svaki tip podatka u Javi, postoji odgovarajući tip u nativnoj funkciji
- Svi primitivni tipovi imaju odgovarajući tip
- Izvestan broj Java klasa (koje dolaze uz VM) imaju odgovarajući tip
- Ako za neku Java klasu ne postoji odgovarajući tip, ona je predstavljena **jobject** tipom podatka
  - Odgovara Object klasi u Javi, koju implicitno nasleđuju sve klase

Java tipovi u JNI

Java primitivni tip	JNI native tip	Opis
boolean	jboolean	unsigned 8 bita
byte	jbyte	signed 8 bita
char	jchar	unsigned 16 bita
short	jshort	signed 16 bita
int	jint	signed 32 bita
long	jlong	signed 64 bita
float	jfloat	32 bita
double	jdouble	64 bita
void	void	-

Java klasa	JNI native tip
java.lang.Object	jobject
java.lang.Class	jclass
java.lang.String	jstring
java.lang.Throwable	jthrowable
niz objekata	jobjectarray
niz boolean primitiva	jbooleanarray
niz byte primitiva	jbytearray
niz char primitiva	jchararray
niz short primitiva	jshortarray
niz int primitiva	jintarray
niz long primitiva	jlongarray
niz float primitiva	jfloatarray
niz double primitiva	jdoublearray

Poziv nativne metode

- Prototip nativne funkcije:

```
JNIEXPORT void JNICALL Java_ClassName_MethodName(JNIEnv *env, jobject jThis)
```

- Minimalan broj argumenata:
  - JNIEnv *env
    - veza sa virtualnom mašinom
    - sadrži veliki broj funkcija virtualne mašine
    - validan samo u okviru poziva metode
  - jobject jThis
    - predstavlja referencu na objekat klase u kojoj se nalazi native metoda (ekvivalent this)

JNIEnv klasa

- Sadrži veliki broj funkcija virtualne mašine, kao i pomoćnih funkcija:
  - String funkcije:
    - NewStringUTF() – kreira Java String; ne koristi se u nativnoj fuknciji, već se, recimo, vraća u Java program
    - GetStringUTFChars() – konvertuje Java string u niz UTF8 karaktera
    - ReleaseStringUTFChars() – oslobađa memoriju zazetu GetStringUTFChars() funkcijom
  - Class-Loader funkcije:
    - FindClass() – učitava zadatu klasu u VM; klasa je zadata imenom paketa i klase, odvojenih kosom crtom:

```
env->FindClass ("com/rtrk/jni/TestJNIActivity");
```

    - GetObjectClass() – vraća klasu zadatog objekta

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

  - Alokacija/dealokacija memorije
    - Get*Tip*ArrayElements() – konvertuje Java niz u C (C++) niz
    - Release*Tip*ArrayElements() – dealocira C (C++) niz zauzet gornjom funkcijom

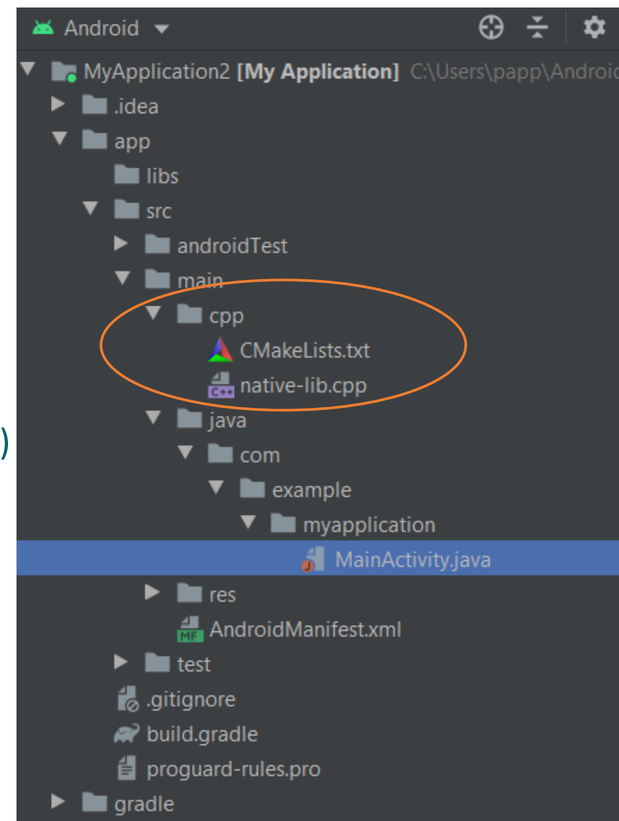
Implementacija nativne funkcije

- C ili C++ fajl uključuje zaglavlje

```
#include <string.h>
#include "com_rtrk_jni_MojJNITestActivity.h‟

JNIEXPORT jstring JNICALL Java_com_rtrk_jni_MojJNITestActivity_stringFromJNI
  (JNIEnv *env, jobject jThis) {
     return env->NewStringUTF("Hello from JNI !");
}
```

Android Studio podrška JNI

- Android Studio zna da generiše projekat sa JNI podrškom
- Generiše dodatni folder cpp sa nativnim kodom
- Ovaj folder će sadržati JNI projekat u kojem će biti implementacija
- cpp folder sadrži:
  - **.cpp** datoteku sa implementacijom nativne funkcije (ili funkcija)
  - **CMakeLists.txt** datoteku koja predstavlja ekvivalent *makefile* datoteke (sadrži uputstvo kako se prevodi i linkuje ovaj projekat)
  - Android Studio prepoznaje vezu između definicije nativne funkcije i Java klase (ukoliko je definicija ispravna)
- Nativni deo koda se automatski uključuje u build



Generisanje prototipova – Android

- Deklariše se nativna metoda u Java klasi npr. MainActivity (primer iz Android Studio):

```
public native String getStringFromJNI();
```

- U static delu koda se učitava biblioteka:

```
static { System.loadLibrary("test-jni"); }
```

  - primetiti da ime biblioteke nema “lib” u nazivu, iako sama datoteka ima “lib” u nazivu
  - Zašto u static?

- Iz konzole shell-a u java folderu startuje se **javah** program:

```
javah com.example.myapplication.MainActivity
```

- Program **javah** generiše zaglavlje (*.h) i taj fajl se premešta iz **java** foldera u **cpp** folder projekta

Rezime

- JNI nije specifičan samo za Android
- Može da se koristi u svakom Java programu
- Treba imati u vidu
  - Prednosti – bolje performanse
  - Mane – prenosivost, više koda, složeniji program
- U mnogim ne-nativnim programskim jezicima postoje slični mehanizmi za proširenje
  - Python
  - JavaScript
  - Lua

Dodatak

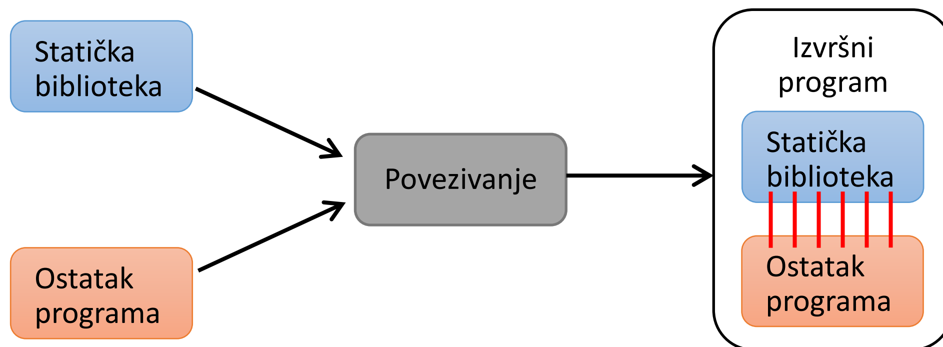
Dodatni detalji JNI

Tipovi programskih biblioteka

- Programske biblioteke po pravilu korespondiraju modulima u arhitekturi programske podrške
- Generišu se na osnovu izvornog koda, upotrebom alata za razvoj (isto kao i izvršni programi)
- Eksportuju kontrolisani skup simbola (funkcija i podataka) koji se mogu koristiti u drugim modulima
- Dve vrste biblioteka
  - Statičke biblioteke
    - Windows – .LIB, Linux, Archive - .A
  - Dinamičke biblioteke
    - Windows, Dynamic Link Library – .DLL
    - Linux, Shared Object - .SO

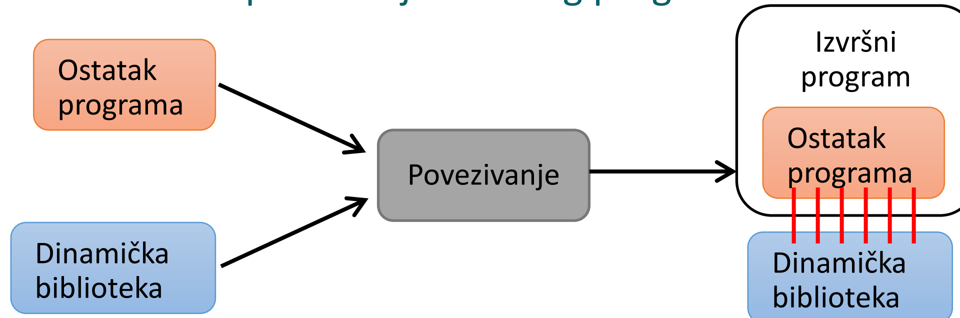
Statičke biblioteke

- Binarni oblik biblioteke je moguće distribuirati samo uz odgovarajuće (npr. .h) zaglavlje u kojem se nalaze deklaracije eksportovanih simbola statičke biblioteke
- Definicije simbola su unutar statičke biblioteke
- Simboli se razrešavaju tokom povezivanja
- Statička biblioteka postaje deo izvršnog programa



Dinamičke biblioteke

- Binarni oblik biblioteke je moguće distribuirati samo uz odgovarajući (npr. .h) header u kojem se nalaze deklaracije eksportovanih simbola dinamičke biblioteke
- Definicije simbola su unutar dinamičke biblioteke
- Simboli se razrešavaju tokom pokretanje izvršnog programa
- Dinamička biblioteka ne postaje deo izvršnog programa, ali je potrebna za ispravan rad istog
- Zavisnost se proverava tokom pokretanja izvršnog programa



Pristup atributima

- Za čitanje vrednosti atributa koristi se funkcija **`Get`*`Tip`*`Field(object, field_id)`**
  - Tip je ili Void, ili neki primitivan tip (npr. Int), ili referenca na objekat neke klase (Object)
  - prvi argument je referenca na objekat čiji atribut čitamo
  - drugi argument je ID atributa
- Za postavljanje nove vrednosti atributa, koristi se funkcija **`Set`*`Tip`*`Field(object, field_id, nova_vrednost)`**
- ID atributa se dobija pozivom funkcije **`GetFieldID(klasa, ime, potpis)`**
  - prvi argument je klasa objekta (reprezentovana jclass instancom)
  - drugi argument je naziv atributa
  - treći argument je potpis
- Klasa objekta se dobija pozivom funkcije **`GetObjectClass(obj)`**
  - argument je referenca na objekat čiju klasu želimo da saznamo

Pristup atributima

- Java

```
// Stvorimo primerak klase, koji se prosledjuje native metodi.
SomeJavaClass sjc = new SomeJavaClass();
// Poziv native metode koja ce promeniti atribut attr.
app.workWithAttr(sjc);
```

- C++

```
JNIEXPORT void JNICALL Java_com_rtrk_jni_TestJNIActivity_workWithAttr
  (JNIEnv *jEnv, jobject jThis, jobject jObj) {
  const char *text = "novi tekst";
  // Pokupimo klasu argumenta funkcije.
  jclass cls = jEnv->GetObjectClass(jObj);
  // Pokupimo ID atributa prosledjenog objekta.
  jfieldID fid = jEnv->GetFieldID(cls, "attr", "Ljava/lang/String;");
  // Kreiramo java.lang.String od niza karaktera (C string).
  // To ce biti nova vrednost atributa attr prosledjene klase.
  jstring jText = jEnv->NewStringUTF(text);
  // Promenimo vrednost atributa na osnovu njegovog ID-a i nove vrednosti.
  jEnv->SetObjectField(jObj, fid, jText);
}
```

Pristup statičkom atributu

- Ne koristi se referenca na objekat, već samo informacija o klasi
- Nazivi funkcija su slični – imaju ubačenu reč “Static”
- Funkcija **`GetStatic`*<u>`Tip`</u>*`Field(klasa, atribut_id)`** vraća vrednost statičkog atributa
- Funkcija **`SetStatic`*<u>`Tip`</u>*`Field(klasa, atribut_id, nova_vrednost)`** postavlja novu vrednost statičkog atributa
- Funkcija **`GetStaticFieldID(klasa, naziv, potpis)`** vraća ID statičkog atributa

Poziv metode

- Metoda se poziva tako što se prvo identifikuje, pa se onda pozove metodom **`CallTipMethod(objekat, metod_id, argument_ili_argumenti)`**
  - Tip je ili Void, ili jedan od primitivnih tipova, ili referenca na neku od klasa (Object)
  - prvi argument je referenca na objekat
  - drugi argument je ID metode
  - treći (četvrti, peti,...) argument je argument metode
- ID metode se dobija pozivom funkcije **`GetMethodID(klasa, ime, potpis)`**
  - prvi argument je klasa objekta čiju metodu pozivamo
  - drugi argument je ime metode
  - treći argument je potpis

Pristup atributima Java klasa

- JNI omogućuje da se iz nativne funkcije pristupa atributima proizvoljne Java klase
- Potrebno je identifikovati klasu, pristupiti objektu te klase (ako atribut nije statički) i identifikovati atribut
- Obično se referenca na objekat prosleđuje kao parametar nativne funkcije
- Atributi se identifikuju imenom i potpisom
  - potpis zavisi od toga da li je atribut:
    - primitivan tip
    - referenca na objekat neke klase

Potpis atributa

- Potpis atributa zavisi od tipa, i od toga da li je niz
- Ako je atribut primitivan tip, koristi se tekst iz tabele dole
- Ako je atribut referenca na objekat neke klase, opisuje se po pravilu za potpis klase (sledeći slajd)

Primitivan tip	Opis
boolean	Z
byte	B
char	C
double	D
float	F
int	I
long	J
short	S

Potpis klasa i nizova

- Ime klase počinje velikim latiničnim slovom L (na primer: LMojaKlasa;),
- Potreban je pun naziv klase koji uključuje i naziv paketa kome pripada
  - naziv paketa se navodi pre imena klase, sa znakom / kao delimiterom unutar paketa (na primer: Ljava/lang/String;),
- Na kraju opisa se nalazi znak ';'
- Nizovi:
  - ako je atribut niz, dodaje mu se u naziv i znak '[' (na primer: **[I** za niz int vrednosti),
  - broj dimenzija niza se zamenjuje brojem znakova '[' (na primer, dvodimenzionalni niz se opisuje znacima '[[')

Dobavljanje JNIEnv

- Da bi nativna metoda pozvala asinhrono neku Java metodu, potrebno je da:
  - pronađe Java klasu,
  - da dobavi referencu na objekat te klase (ako nije statička metoda),
  - da identifikuje metodu (dobavi njen ID) i
  - da je pozove
- Za sve navedene operacije, potreban je JNIEnv objekat
- Do njega se dolazi indirektno:
  - prilikom inicijalizacije dinamičke biblioteke, JNI podsistem poziva funkciju (ako postoji):

```
jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
```

JNI_OnLoad funkcija

- Ako je napišemo, sistem će je pozvati
- Prvi argument je referenca na VM:
  - **`JavaVM *vm;`**
  - njega zapamtimo u eksternoj promenljivoj, da bi i druge nativne funkcije mogle da koriste VM
- Klasa JavaVM poseduje metodu:

```
GetEnv((void **)&env,JNI_VERSION_1_4)
```

  - upisuje referencu na JNIEnv u prvi argument
  - drugi argument je minimalna verzija Jave
  - vraća:
    - konstantu JNI_OK, ako je sve u redu
    - konstantu JNI_EDETACHED, ako programska nit iz koje se poziva nije deo Java VM (i tada je potrebno da se zakači za Java VM)
    - konstantu JNI_EVERSION, ako nije podržana za zadatu verziju Jave
- Funkcija GetEnv() unutar JNI_OnLoad funkcije ne vraća JNI_EDETACHED, pošto se poziva iz glavne niti Java aplikacije

JNI_OnLoad funkcija

- Ako je sve u redu, vraća konstantu JNI_VERSION_1_4
- Ako postoji neka greška, vraća -1
- Unutar ove metode se zapamti referenca na VM, i može da se potraži Java klasa čiju metodu ćemo pozvati, kao i ID metode koju ćemo pozvati
- Klasa čiju metodu ćemo pozvati iz nativne funkcije mora da se zapamti kao globalna referenca
  - prilikom poziva nativne funkcije iz jave, sve prosleđene reference su lokalne (java metoda → nativna funkcija)
  - funkcija NewGlobalRef(lokalna_referenca) vraća globalnu verziju prosleđene lokalne reference
    - dostupna iz svih nativnih funkcija
  - funkcija DeleteGlobalRef(globalna_referenca) briše globalnu refefrencu i oslobađa memoriju

JNI_OnLoad funkcija

```
static jobject mObject;// Global reference to object of TestJNIActivity class
static jclass mClass;  // Global reference to TestJNIActivity class
jmethodID toastMid;    // callback to Android (Method ID of a toast() method in TestJNIActivity class)
static JavaVM *jVM = NULL;  // Java VM

jint JNICALL JNI_OnLoad(JavaVM* vm, void* reserved)
{
    JNIEnv *env;
    jVM = vm;
    if (vm->GetEnv((void **)&env,JNI_VERSION_1_4)!=JNI_OK) {
        return -1;
    }
    mClass = env->FindClass ("com/rtrk/jni/TestJNIActivity");
    if(mClass == NULL){
        return -1;
    }
    mClass = (jclass) env->NewGlobalRef(mClass);
    mObject = NULL;
    toastMid = env->GetMethodID(mClass,"toast","()V");
    if(toastMid == NULL) {
        return -1;
    }
    return JNI_VERSION_1_4;
}
```

Pamćenje reference na objekat

- Ako se poziva metoda koja nije statička, potreban je objekat, čiju metodu pozivamo
- To se radi iz obične, sinhrone JNI native funkcije, samo se referenca sačuva kao globalna
  - Java metod → nativna funkcija
  - tu postoji kontekst i JNIEnv objekat

```
JNIEXPORT void JNICALL Java_com_rtrk_jni_TestJNIActivity_workWithCallback
  (JNIEnv *jEnv, jobject jThis)
{
  mObject = jEnv->NewGlobalRef(jThis);
}
```

Poziv callback metode iz nativne funkcije

- Nativna funkcija koja će pozvati Java callback metodu mora prvo da dobije JNIEnv
  - za to koristi JavaVM klasu, čiji objekat smo sačuvali kao globalnu promenljivu
- Ako funkcija GetEnv() vrati vrednost JNI_EDETACHED , potrebno je da se trenutna (nativna) nit zakači na Java VM, pozove Java metodu i otkači od Java VM
  - to radi funkcijama:
    - `jVM->AttachCurrentThread(&env, NULL)` i
    - `jVM->DetachCurrentThread()`

Dobavljanje JNIEnv, asinhrono

```
jint res = jVM->GetEnv((void **)&env,JNI_VERSION_1_4);
if (env == NULL)
{
  return NULL;
}
if (res == JNI_EDETACHED)
{
  attached = jVM->AttachCurrentThread(&env, NULL);
  if(attached != 0)
  {
    __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "Failed to attach current thread");
    return NULL;
  }
}
else if (res == JNI_EVERSION)
{
  __android_log_print(ANDROID_LOG_DEBUG, DEBUG_TAG, "The specified interface is not supported");
  return NULL;
}
```

Callback

- Ako pozovemo nativnu metodu, a ona pozove Java metodu, i to se može smatrati callback pozivom
  - to je sinhroni poziv callback metode, gde je kontekst poziva definisan unutar poziva nativne funkcije
  - Java metoda→ nativna funkcija →druga Java metoda
  - nativna funkcija koristi JNIEnv klasu (prvi argument) za identifikaciju i poziv druge Java metode
- Problem može biti asinhroni callback, gde ne postoji kontekst iz kojeg bi bila pozvana Java metoda:
  - nativna funkcija → Java metoda
  - nativna funkcija nije pozvana iz Jave, pa nema JNIEnv klasu, koja predstavlja vezu sa Java VM

Asinhroni poziv Java callback metode

```
env->CallVoidMethod(mObject, toastMid);
env->DeleteGlobalRef(mObject);
mObject = NULL;

if (attached == 0)
{
  jVM->DetachCurrentThread();
}
```

Asinhroni poziv statičke callback metode

- Jednostavnije od prethodnog slučaja, pošto nije potrebno da se zapamti referenca na objekat – za poziv statičke metode je dovoljna samo klasa
- Funkcija JNI_OnLoad je slična, samo se ID metode dobija drugačije:

```
toastMid = env->GetStaticMethodID(mClass,"toast","()V");
```

- Poziv je jednostavniji:

```
env->CallStaticVoidMethod(mClass, toastMid);

if (attached == 0)
{
  jVM->DetachCurrentThread();
}
```