27. Fejezet

Kapcsolatok a helyiekkel

Egész könyvünk arról szól, hogy bebizonyítsuk a következő állításokat:

1. Java nyelven mindent meg lehet írni.
2. Ha valamit nem lehet megírni, akkor alkalmazd az első pontot!

Amit mégsem lehetne Java nyelven megírni, azt az adott platformhoz illeszkedő *natív*[1] interfészen keresztül a Java programhoz kapcsolhatjuk, így téve kerekké a fenti bizonyítást. És hogy ez pontosan hogyan történik, arról szól ez a fejezet...

27.1. Egy kis magyarázkodás

Hogy miért is vetemedik valaki arra, hogy elhagyja a Java nyújtotta környezetet, annak több oka is lehet.

A legegyszerűbb esetben a régi bevált könyvtárait idő-, vagy pénzhiány miatt nem tudja egyik pillanatról a másikra 100%-osan Java nyelven írt osztályokkal felváltani, de mégis át szeretne térni az új platformra. Ilyen esetben a régi könyvtárakat egy szabványos protokollon keresztül is elérheti, de ez egyetlen monolitikus alkalmazásnál nem megoldás. Ilyenkor a Java program részévé kell tenni a régi kódot, amire a natív interfész a legegyszerűbb megoldás.

Ha valaki már írt számításigényes alkalmazást, és ezt az elmúlt években véletlenül Java nyelven is megpróbálta, az szomorúan tapasztalhatta, hogy minden jó szándéka ellenére a programja itt lassabban futott. Sajnos, a virtuális gépben a bájtkód részben interpretált voltát (lásd Q.) már csak a biztonsági követelmények betartatása miatt sem lehet teljesen kiküszöbölni, ezért a program – az egyre hatékonyabb virtuális gépek megjelenése ellenére – lassabban fut. Ilyenkor is jól jöhet, ha a kritikus részeket egy másik nyelven megírva gyorsíthatjuk a program futását.

Harmadik esetként – az interfész alkalmazásának indoklásakor – azt lehet felmutatni, hogy nincs (még) minden megírva Javában, ezért néha más programozók által más nyelven írt könyvtárakat is használnunk kell. Ebben is csak az segíthet, ha azt a natív interfészen keresztül illetsztjük programunkhoz (hacsak magunk meg nem írjuk).

27.2. Egy rövid példa

Hogy hűek maradjunk az új programozási eszközök bemutatása során bevált szokásokhoz, de azért mégis mutassunk valami újat, most a könyv első fejezetében látható HelloVilag alkalmazás angol nyelvű változatát láthatjuk.

Ebben a példaprogramban a tényleges munkát végző részt C nyelven írjuk meg, mondjuk „a gyorsaság kedvéért”.

1. A Java nyelvű kódrészlet:

```
class HelloWorld {
    public native void displayHelloWorld();
```

[1] A „natív” szó itt az adott platformon futtatható bináris kódra utal.

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

    public static void main(String[] args) {
        new HelloWorld().displayHelloWorld();
    }
}
```

A példában két fontos újdonság van:

- A native módosítóval ellátott metódusnak nincs törzse, ezt ugyanis C nyelven implementáljuk. A futtató környezet az ilyen metódusokhoz a natív eljárásokat tartalmazó könyvtár betöltésekor rendeli hozzá a megfelelő törzseket.
- A loadLibrary rendszerhívás betölti a megadott nevű natív könyvtárat, melynek helye és teljes neve platformtól függően változhat (tipikus a .so, avagy a .dll végződés; keresés az LD_LIBRARY_PATH, avagy a PATH által meghatározott könyvtárakban).

2. Fordítsuk le a példaprogramot:
   ```
   javac HelloWorld.java
   ```
3. majd generáljunk belőle egy C header állományt, melyet akár kiindulópontnak is használhatunk az implementáció elkészítéséhez, ugyanis tartalmazza a megvalósítandó függvény(ek) deklarációját:
   ```
   javah -jni HelloWorld
   ```
   A törzset a következő C forrás tartalmazza:
   ```
   #include <jni.h>
   #include "HelloWorld.h"
   #include <stdio.h>

   JNIEXPORT void JNICALL
   Java_HelloWorld_displayHelloWorld(JNIEnv *env, jobject obj)
   {
       printf("Hello world!\n");
   }
   ```
4. A C forrásból már csak natív könyvtárat kell gyártani.

   4w A *Win32-es* platformon a következő – könnyed – parancs használható:
   ```
   cl -nologo -MT -I%JDKHOME%\include -I%JDKHOME%\include\win32
     -LD HelloWorld.c -FeHelloWorld.dll -link
     -libpath:%DEVHOME%\VC\lib -libpath:%JDKHOME%\lib
   ```
   A %JDKHOME% környezeti változónak a JDK könyvtárára, a %DEVHOME% változónak a *Microsoft Visual C++* könyvtárára kell mutatnia.

   4c Ugyanezt a lépést *Cygnus GNU Win32* környezetben a következő parancsok használatával tehetjük meg:
   ```
   gcc -DBUILDING_DLL=1 -D_DLL=1 -I$(JDKHOME)/include
     -I$(JDKHOME)/include/win32 -c HelloWorld.c
   ddlwrap --output-def HelloWorld.def --add-stdcall-alias
     -o HelloWorld.dll HelloWorld.o
     -Wl,-e,__cygwin_noncygwin_dll_entry@12 -s
   ```
   A $JDKHOME környezeti változó itt is a JDK könyvtárára mutat[2]

[2] Cygwin b20.1, egcs-1.1.2 verziókkal és egy módosított $JDKHOME/include/win32/jni_md.h fájllal működik.

4s *Solaris* operációs rendszer alatt a következő parancsot kell kiadni:

```
cc -G -I. -I$(JDKHOME)/include -I$(JDKHOME)/include/solaris
  HelloWorld.c -o libHelloWorld.so
```

4l És végül – de nem utolsó sorban – *Linux*[3] alatt a következő paranccsal oldható meg a fordítás:

```
gcc -shared -I. -I$(JDKHOME)/include -I$(JDKHOME)/include/linux
  HelloWorld.c -o libHelloWorld.so
```

5. Munkánk gyümölcsét a következő parancs beírása után élvezhetjük:

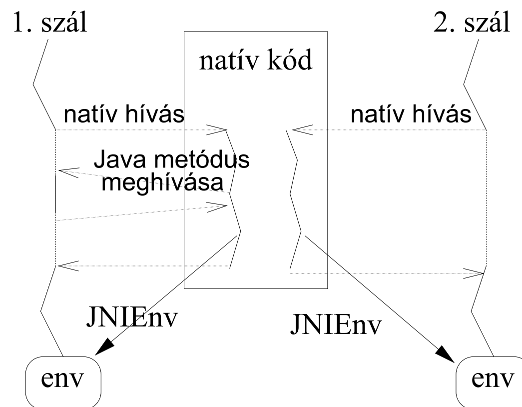
```
java HelloWorld
```

Ha a képernyőn megjelent a „Hello world!" felirat, akkor megengedhetünk magunknak egy kávészünetet, mert a nehezén túl vagyunk. Ezek után már „csak" a függvényhívások ismertetése következik...

...ha nem sikerült, akkor először nézzük meg, hogy a %PATH%, illetve LD_LIBRARY_PATH környezeti változók alapján rá lehet-e találni a lefordított osztott könyvtárra ...ha ez sem segít, akkor a virtuális gépünk nyomkövető funkcióival kell megismerkednünk.

27.3. Felépítés

A hivatalosan kiadott specifikáció csak C és C++ nyelvekhez ad illesztési felületet, de ezekre alapozva más nyelvek illesztése is elkészíthető. A következőkben C nyelvű példákat és szerkezeteket mutatunk, ezért e nyelv ismerete szükséges a megértéshez.



27.1. ábra: Natív eljárások hívása

27.3.1. Működés

A Java nyelv oldaláról a natív metódusok hívása nem tér el a szokásos metódushívásoktól, a különbségeket a virtuális gépnek kell elfednie. A native módosítószóval ellátott metódus törzse platformspecifikus kódban van megvalósítva. Ezt a törzset a *natív metódus hívásakor* aktivizálja a virtuális gép a C nyelv hívási szemantikájának megfelelően.

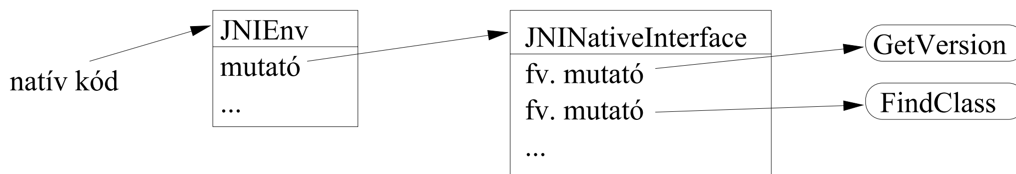
[3] JDK 1.2 pre-beta v2

Mivel a C nyelv sem az objektumorientált, sem a többszálú programok írását nem támogatja nyelvi szinten, ezért ezen specialitások kezelésére a Java kód aktuális állapotát leíró szerkezetek használata szükséges (lásd 27.1. ábra). Ez az állapot szálanként eltérő lehet, ezért mindegyik szálhoz egy-egy különálló *állapotleíró struktúra* (az ábrán env) tartozik.

Amikor a natív kód Java nyelvi elemeket szeretne használni, akkor azt ezen a környezetleíró szerkezeten keresztül teheti csak meg. A natív kód így a Java nyelv minden elemét elérheti: adattagokat kérdezhet le vagy módosíthat; *metódusokat hívhat* meg; objektumokat hozhat létre; szálakat indíthat el vagy kezelhet; stb.

27.3.2. Szerkezet

Minden natív metódus első paraméteréül az állapotleíró struktúrára hivatkozó mutatót kapja. A hivatkozott szerkezet legfontosabb eleme a felület lényegét adó függvényhívások gyűjteményére hivatkozó mutató (JNINativeInterface). Ez a gyűjtemény lényegében a C++ nyelv virtuális metódustáblájának felel meg.



27.2. ábra: A felület felépítése

A felület egy függvényének használata C nyelvben tehát a következő:

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

C++-ban ezt egy picit egyszerűbben is le lehet írni:

```
env->GetVersion();
```

Egy ilyen gyűjtemény használata lehetővé teszi, hogy az egyes szálak ugyanannak a függvénynek más-más implementációját érjék el. Gyors, nyomkövetési kódot vagy biztonsági ellenőrzéseket tartalmazó verziója is lehet ugyanannak a függvénytörzsnek.

Egy szálból hívott natív kódú programrészletek azonos függvénygyűjteményt fognak látni, azonban más-más szálakban ez eltérő lehet.

Ha a Java metódus nem osztályszintű, akkor a függvény második paramétere az aktuális objektumra hivatkozó referencia lesz.

27.3.3. Betöltés és illesztés

A natív metódusok törzsét tartalmazó kódot a System.loadLibrary metódussal lehet inicializálni. Ez a metódushívás a paraméterül kapott nevet platformspecifikus alakra hozza, majd az aktuális ClassLoader segítségével megpróbálja betölteni. Ha ez sikerül, akkor a kódot a virtuális géphez illeszti és a Java kódban lévő natív metódushívásokat az őket implementáló kódrészletekkel összekapcsolja.

Hogy a folyamatról fogalmat alkothassunk, adjuk ki a következő parancsot:

```
java -verbose:jni HelloWorld
```

A Java futtató rendszer a natív kód beépítésére az úgynevezett *osztott könyvtárak* technikáját használja. Ennek az a lényege, hogy a futtatható kód nem minden darabját építi bele magába a programba, hanem a közös részeket közösen használt fájlokban helyezi el.

Ezekre a fájlokra név szerinti hivatkozás van a program kódjában, így ha annak szüksége van egyikükre, akkor dinamikusan használja fel: megkeresi a fájlrendszerben, betölti, majd a megfelelő darabját lefuttatja. Ezek a fájlok általában függvényeket tartalmaznak, melyekre szintén név szerint lehet hivatkozni.

Azt a folyamatot, melynek során a név szerint megnevezett könyvtárból egy megnevezett függvény meghívásra kerül, *dinamikus kötésnek* nevezzük.[4] A dinamizmus két ponton érzékelhető:

- Az osztott könyvtárak bármikor lecserélhetőek, így az összes őket használó program felfrissül.
- A fájl keresése és betöltése elmaradhat, ha azt már más program behívta a memóriába. Ezzel időt és helyet is megtakaríthatunk.

Az osztott könyvtár betöltését a következő metódushívással kezdeményezhetjük:

```
System.loadLibrary("HelloWorld");
```

Az első példában látott kódrészlet a HelloWorld nevet egészíti ki UNIX rendszerekben általában libHelloWorld.so, Win32-es platformon pedig HelloWorld.dll alakra.

Ha a platformspecifikus néven elérhető a kód, akkor azt az aktuális ClassLoader betölti. Ha a betöltött kódban a JNI_OnLoad függvény definiálva van, akkor ezt a virtuális gép meghívja, így annak törzse a natív kódban lévő struktúrák kezdeti inicializálására is felhasználható. Ennek párját, a JNI_OnUnload függvényt a natív metódusokat deklaráló osztály(ok) felszabadítása után hívja meg a virtuális gép.

Egy ilyen függvénykönyvtár több osztályhoz tartozó natív metódusokat is megvalósíthat, de csak akkor illeszthető mindegyikhez, ha azonos betöltővel (ClassLoader) lettek a virtuális gépbe betöltve.

Ha az adott platform nem támogatja függvénykönyvtárak dinamikus betöltését, akkor azokat a virtuális géppel statikusan kell összekapcsolni[5]. Ebben az esetben a loadLibrary metódushívás nem csinál semmit.

A deklarált metódusok és az őket megvalósító törzsek összeillesztése név alapján automatikusan megtörténik, de a programozó ezt is a kezébe veheti a RegisterNatives függvény használatával[6]. Ha a metódus neve nem túlterhelt, akkor az automatikus illesztés csak a metódus neve alapján, ha túlterhelt, akkor *formális paraméterlistájának* (szignatúra) felhasználásával történik meg.

Az automatikus megfeleltetés során haszált nevek elkészítésére a "javah -jni" parancs használatát javasoljuk, mert a generálási szabályok túl bonyolultak kézi használatra.

Metódusnevek megfeleltetése

A metódusnevek a következő elemekből állnak össze:

- a Java_ prefix
- egy kicsit átgyúrt (lásd később), csomagokkal minősített osztálynév
- egy „_” választó karakter
- egy kicsit átgyúrt metódusnév
- túlterhelt metódusoknál az átgyúrt szignatúra két aláhúzásjel után („__”)

Az első példában a név nélküli csomagban lévő HelloWorld osztály displayHelloWorld metódusának megfelelő natív név a következő:

```
Java_HelloWorld_displayHelloWorld
```

[4]A virtuális metódusok kiválasztásának és meghívásának is ez a neve, mert lényegében ugyenez történik, lásd 4.11.2.

[5]Ez általában a virtuális gép újrafordítását jelenti.

[6]Érdekes megfigyelni a -verbose:jni kapcsoló hatására megjelenő sorokban, hogy ez egy ritkán használt fogás.

Az említett *átgyúrás* során a minősítésekben lévő elválasztó karakter („." vagy „/") helyére egy aláhúzásjel kerül („_"). Mivel azonosítók nevei nem kezdődhetnek számjeggyel, ezért a _0, ..., _9 kombinációk speciális jelek kódolására szolgálhatnak:

kód	**jelentés**
_0XXXX	az XXXX kódú Unicode karakter
_1	az „_" karakter
_2	a „;" karakter szignatúrákban
_3	a „[" karakter szignatúrákban

Paraméterek megfeleltetése

Amennyiben egy metódus neve túlterhelt, akkor a natív metódusnévbe formális paraméterlistájának típusai is belekerülnek. Erre legyen példa a következő metódus:

```
package greeting;
public class Hello {
    native void multi(int i, String text);
    ...
}
```

A metódus nevének hosszú változata, ekkor a következő:

`Java_greeting_Hello_multi_ILjava_lang_String_2`

A szignatúra kódolásának első lépésében a formális paraméterlista típusai módosulnak a következő táblázatnak megfelelően:

kódolt alak	**típus**
B	byte
C	char
D	double
F	float
I	int
J	long
Losztálynév;	osztály vagy interfész
S	short
Z	boolean

Tömb esetén az elemek típusát – minden dimenziót reprezentálandó – egy-egy „[" karakter előzi meg (lásd a java.lang.Class osztályt a referenciában).

Az így kapott karaktersorozatot a metódusneveknél látott szabályok szerint a C nyelv szintatkikájának megfelelő alakra alakítja át a rendszer.

27.3.4. Hivatkozás objektumokra

Primitív típusú változók érték szerint (másolva) kerülnek átadásra a Java és a natív kód között. Objektumok ezzel szemben referencia szerint, ezért ezek sorsát a virtuális gépnek a natív kódban is tudnia kell követni. A natív kódnak átadott és ezáltal használt objektumokat a szemétgyűjtő eljárás automatikusan nem szabadíthatja fel, de a natív kód ezek felszabadítását explicit módon mégis megengedheti.

Referenciák

Három alapvető referencia kategória van a Java nyelvben, melyek a natív felületen keresztül is elérhetőek: globális, lokális és gyenge referenciák. Megfelelő függvényhívások használatával minden objektumhoz készíthető bármilyen referencia és meg is szüntethető az.

A globális referenciák addig maradnak érvényesek, ameddig explicit módon nem törlik őket. Az általuk hivatkozott objektum az explicit felszabadítás idejéig él, ha más hivatkozás már nincsen rá.

Gyenge referenciák a globálisakhoz hasonlóan érvényesek, de az általuk hivatkozott objektumot a szemétgyűjtő eljárás már akkor is megszüntetheti, ha csak gyenge referenciák hivatkoznak rá és szüksége van a helyre (lásd P.).

Lokális referenciák a hozzájuk tartozó blokkban érvényesek és az általuk hivatkozott objektumok élettartama ennek megfelelő, hacsak más referenciával nem hivatkozunk rájuk. Lokális referenciákat más végrehajtási szálnak sem lehet átadni, mert ott már nem érvényesek.

A natív metódusoknak paraméterül átadott és a natív interfész függvényeinek visszatérési értékeként kapott referenciák lokálisak. A felület függvényeiben paraméterként, illetve natív metódusok visszatérési értékeként mindhárom referenciafajta használható.

Szemétgyűjtés

Egy virtuális gépnek minden, a natív kódnak átadott referenciát követnie kell, hogy a hozzájuk tartozó objektumokat ne szabadítsa fel. E cél megvalósítása érdekében minden, a natív kódnak (paraméterként, vagy az interfész függvényének visszatérési értékeként) átadott, referenciát fel kell jegyeznie. A feljegyzésre került, de globálissá nem átalakított referenciákat a natív kód lefutása után meg lehet szüntetni, ezáltal lehetőséget biztosítva a szemétgyűjtésnek az objektum felszabadítására.

A natív kód futása közben konzervatív szemétgyűjtési eljárással[7] az objektumok nem szabadíthatóak fel, mert referenciákat globális vagy dinamikusan lefoglalt memóriaterületeken is lehet tárolni. Natív kód futása közben csak azok a referenciák tekinthetőek érvénytelennek, s törölhetőek a fent említett feljegyzésekből, melyeket a program explicit módon megszüntet (lásd DeleteLocalRef).

Lokális változók

Általában a natív kód a virtuális gépre hagyatkozhat a referenciák kezelésében, azonban lokális referenciák átmeneti használata sok fölösleges memória lefoglalásához vezethet. Ezek egy része a vermen, más része a dinamikusan kezelt területeken helyezkedik el:

- A natív kód rövid ideig nagy struktúrákat használhat, melyek a futás további lépéseiben már nem szükségesek. Ebben az esetben a dinamikusan kezelt memóriaterületeken nagy részek lesznek használhatatlanok a program számára, holott azokat egy szemétgyűjtési eljárás felszabadíthatná.
- A natív kód sok kis objektumot használhat, melyek mindegyikéhez egy lokális referenciát hozhat létre (például egy tömb elemein egy ciklus végigmegy). A kód ezek mindegyikét csak rövid ideig használja, ennek ellenére a vermen mindegyik helyet fog elfoglalni, ha nincs explicit módon felszabadítva. (A verem telítettsége az EnsureLocalCapacity fügvénnyel ellenőrizhető.)

Mindkét esetben a már nem használt lokális referenciák felszabadítása segíthet, mely történhet explicit módon (DeleteLocalRef), vagy egy blokk kezdetének és végének megjelölésével (PushLocalFrame és PopLocalFrame). Utóbbi esetben a blokk végén a megjelölt kezdőpont után létrehozott lokális referenciák szűnnek meg.

[7] Erősen leegyszerűsítve: a vermet végignézve megjelöljük a hivatkozott blokkokat, majd a meg nem jelölteket felszabadítjuk.

27.3.5. Elemek elérése

A natív interfész objektumok kezeléséhez függvények gazdag választékát nyújtja, melyek a platformtól függetlenül oldják meg az adatok kezelését. Ez a függetlenség az egyes esetekben elkerülhetetlen konverziók miatt lassulást okozhat, ám lehetővé teszi a natív kód tökéletes integrálását tetszőleges virtuális géppel. A natív interfész egyes esetekben lehetővé teszi a belső ábrázolás elérését, ám csak igen erős megkötésekkel (rövid és gyors kell, hogy legyen, mert a többi végrehajtási szál ilyenkor leáll).

Primitív tömbök kezelése

A tömbelemek kezelése olyan terület, ahol bármilyen kis időveszteség jelentősen csökkentheti egy alkalmazás hatékonyságát. A kis időveszteségek tömbelemek iterálásakor (vektor vagy mátrix műveletek) összeadódnak, és így már számottevően nagy veszteség lesz belőlük.

A probléma egyik megoldása, hogy az ilyen tömbökhöz megengedjük a közvetlen hozzáférést. Ilyenkor a natív kód egy mutatót kapna a tömbelemekre. A megoldás követelményei:

- A szemétgyűjtő eljárásnak támogatnia kell a közvetlen hozzáférést is.
- A virtuális gépnek az adott platformnak megfelelően, folyamatosan kell elhelyeznie a tömböt a memóriában. Noha a megvalósítások nagy részében ez a követelmény természetesen teljesül, lehetnek apróbb eltérések (pl. a logikai tömbök elemeit tömör formában biteken is lehet ábrázolni).

A natív interfész olyan köztes megoldást kínál, amely ezeket a lehetséges buktatókat kikerüli.

Az interfész megfelelő függvényeivel egy primitív típusokból álló Java tömb egésze (Get<Típus>ArrayElements), vagy részei (Get<Típus>ArrayRegion) a natív kód számára hozzáférhetővé válnak. Ha a virtuális gép a fenti két követelményt teljesíti, akkor a tömbhöz közvetlen hozzáférést is megengedhet. Ha nem teljesíti, akkor a művelet eredményeképpen a megfelelő elemekről egy lokális másolat készül.

A felület megfelelő függvényeivel (Release<Típus>ArrayElements) lehet jelezni a virtuális gép számára a módosítások végét, ha a natív kódnak már nincs szüksége a tömbelemekre, vagy a Java kódnak is tudomására szeretné hozni a rajtuk végzett változtatásokat. Közvetlen hozzáférés esetén nem történik semmi, de ha egy lokális másolat készült, akkor az visszamásolódik az eredeti tömbbe.

Az itt vázolt esetekben a virtuális gép minden tömb elérésekor eldöntheti, hogy melyik módszert választja. Elképzelhető az is, hogy egy már lemásolt tömbhöz, más helyekről közvetlen hozzáférést is biztosít.

Olyan kódrészleteknél, ahol a gyorsaság miatt csak a közvetlen hozzáférés elfogadható, kivételes helyzetekre fenntartott elérési módszer ([Get|Release]PrimitiveArrayCritical) is használható. Ezen elérési módszer elég erős megkötésekkel jár:

- A hozzáférés ideje (kritikus szakasz) alatt semmilyen más interfészbeli függvény nem hívható (kivétel egy tömb ugyanilyen elérése).
- A kritikus szakasznak gyorsan véget kell érnie, ugyanis a szemétgyűjtés ez alatt nem indulhat el.

Ugyan a közvetlen hozzáférés itt sem garantált, de valószínűsége sokkal nagyobb a megkötések miatt.

Szövegek kezelése

A szövegek kezelése a primitív tömbökéhez hasonló, a karakterek ábrázolási formájából származó eltérésekkel. A C nyelvben egy karakter csak 8 bites, míg a Java nyelvben 16 bites. A natív kód a szélesebb karaktereket Unicode (16 bites), vagy UTF (Unicode Transfer Format, lásd 12.) formátumban érheti el. Az elérés során, a virtuális gép belső ábrázolási módszerétől függően, konverzió és másolás is történik.

Lehetőség van szövegek közvetlen kezelésére is, a primitív tömbök kritikus szakaszban történő elérésére megadott megkötésekkel.

Adattagok és metódusok kezelése

A natív kód számára adattagok és metódusok is elérhetőek. Az ismételt elérések gyorsítására az azonosítás külön fázisban történik.

Az első lépésben a név és a szignatúra alapján (lásd 11) a virtuális gép megkeresi a megfelelő tagot. A clazz osztály double f(int, String) metódusának megkeresése:

```
jmethodID mid =
  env->GetMethodID(clazz, "f", "(ILjava/lang/String;)D");
```

A keresés eredményeképpen kapott azonosítót lehet adattagok és metódusok elérésére használni, pl.:

```
jdouble result = env->CallDoubleMethod(obj, mid, 10, str);
```

Természetesen osztály szintű metódusok és adattagok is elérhetőek ilyen azonosítókkal:

```
jmethodID smid =
  env->GetMethodID(clazz, "main", "([Ljava/lang/String;)V");
env->CallStaticVoidMethod(clazz, smid, args);
```

Egy mező- vagy metódushivatkozás lekérdezése után a virtuális gép az adott osztályt megszüntetheti, melynek hatására a hivatkozás érvénytelenné válik. A meglepetések elkerülése végett a natív kódnak egy érvényes referenciával kell hivatkoznia a célosztályra (lásd 27.3.4.), vagy mindig újra le kell kérdeznie ezeket a hivatkozásokat, ha hosszabb ideig is használni akarja őket.

A mező- vagy metódushivatkozás megszerzésének egy másik módja az önelemzés során kapott objektumok használata (lásd 18.). A natív interfész lehetőséget biztosít egy ilyen tagot reprezentáló objektum és egy hivatkozás közötti konverzióra ([From|To]Reflected [Method|Field]).

27.3.6. Hibakezelés

A natív interfész függvényei nem végeznek teljes körű ellenőrzést a paraméterül kapott változókon, mert ez egyrészt jelentősen csökkentené hatásfokukat, másrészt nem is mindig lehetséges, hiszen nem állnak rendelkezésre ugyanazok a típusinformációk, mint a Java nyelvben. Érvénytelen vagy hibás paraméterekkel meghívott függvények, a C nyelvhez hasonlóan, a program leállásához, esetleg a virtuális gép megállásához is vezethetnek.

Kivételek és hibakódok

Noha teljes körű ellenőrzés nincsen, de bizonyos ellenőrzés van a különböző függvényekben. Ha egy függvény lefutása a hibás paraméterezés vagy valamilyen más okból nem sikeres, akkor erről a programozó két forrásból értesülhet:

- A függvény abnormális visszatérési értékéből (mint a NULL mutató).
- Kiváltott kivétel ellenőrzéséből (ExceptionCheck).

Ha a függvény visszatérési értékeiből előre nem jelölhető ki abnormális elem (pl. Java metódushívások), akkor csak a kivétel jelezheti a hibás állapotot, de az összes többi esetben a visszatérési érték ellenőrzése elegendő.

Aszinkron kivételek

További nehézséget jelenthet, hogy más szálak is kiválthatnak a natív kódot futtató szálban kivételeket. Ezek csak akkor kerülhetnek felszínre, ha a program explicit módon ellenőrzi azt az ExceptionCheck függvényhívással, vagy a natív interfész egy függvényét hívja meg.

A natív interfész függvényei közül is csak azok ellenőrzik kivételek meglétét, amelyek maguk is kiválthatnak egyet.

Az aszinkron kivételek (például InterruptedException) belátható időn belüli kezelése érdekében a natív kód hosszú ideig tartó és kivételkezelés nélküli részeiben érdemes rendszeresen meghívni a ExceptionCheck függvényt.

Kivételkezelés

A már kiváltott kivételek kezelésére két módszer kínálkozik:

- A natív kód gyorsan befejezi a működését és visszaadja a vezérlést a Java programnak, a kivételkezelés feladatát sikeresen átruházva arra.
- Törli a kivételt (ExceptionClear), majd végrehajtja saját kivételkezelő rutinját.

Ha van kezeletlen kivétel, akkor annak eltüntetéséről gondoskodni kell minden további natív interfészbeli függvényhívás előtt. A natív interfészből persze egyes függvényeket (ExceptionCheck, ExceptionOccured, ExceptionDescribe és ExceptionClear) meg lehet hívni a „kivételes” állapot megszüntetésére.

27.3.7. Szinkronizálás

A Java nyelv synchronized (lásd 15.) módosítója a natív metódusokra is használható, ilyenkor a kölcsönös kizárást a natív függvényt meghívó virtuális gépnek kell megvalósítania.

Rövidebb kritikus szakaszok védelmére a synchronize(o) nyelvi elem használható a Java nyelvben. Mivel ehhez hasonló szerkezet nincs sem a C, sem a C++ nyelvben, ezért az ilyen szakaszok elejét (MonitorEnter) és végét (MonitorExit) explicit függvényhívásokkal kell jelezni.

27.4. Sorolvasás

Ha nincs feltétlenül szükségünk a grafikus felületre, de Java alkalmazásunkban mégis valamilyen módon kapcsolatot kell tartanunk a felhasználóval, akkor elég kevés eszköz áll rendelkezésünkre.

Ezen segíthet egy parancssori felület, amely a legtöbb POSIX kompatibilis operációs rendszerben megtalálható: a *readline* csomag.

A *readline* csomag gazdag szerkesztési és történeti (visszaléphetünk a korábban bevitt sorokra) lehetőségeket kínál, így ideális lenne céljainkra. Az egyetlen probléma vele, hogy C nyelven van megírva.

Most ehhez a csomaghoz készítünk natív felületet, hogy mégis fel tudjuk használni Java alkalmazásainkban, s a fejezet bevezetőjében említett állításoknak eleget tegyünk.

27.4.1. A *readline* csomag

A *readline*[8] csomag következő elemeit szeretnénk felhasználni:

readline: `char *readline(char *prompt)`
Egy sor beolvasása a bemenetről. A prompt-ban megadott szöveg kerül kiírásra a sor elején.

read_history: `int read_history(char *filename)`
A történeti tömb (amelyben vissza lehet lépni korábbi sorokra) a paraméterül adott fájl tartalmával lesz kiegészítve. Ha a fájlnév nincs megadva, akkor a „~/.history"[9] fájlt használja. Hiba esetén 0-tól eltérő értékkel tér vissza, mely a szokásos hibakódokra utal.

write_history: `int write_history(char *filename)`
A történeti tömb kiírása az adott fájlba. Egyébként a read_history-nak megfelelően viselkedik.

add_history: `void add_history(char *line)`
Egy sor hozzáadása a történeti tömbhöz.

Az elemek illusztrálására egy C nyelvű példaprogram szolgál:

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <readline/readline.h>
#include <readline/history.h>

int main(int argc, char *argv[]) {
        char*   line = (char *)NULL;            // beolvasandó sor
        int     toQuit = 0;                     // kilépési feltétel

        read_history("./trl.history");
        while(! toQuit) {
                line = readline("Halihó> ");    // sor beolvasása
                if(! line) break;               // Ctrl-D, avagy EOF
                if(*line) {                     // ha "l" nem üres
                        printf("%s\n", line);
                        add_history(line);      // történethez hozzáfűzés
                        if(strcmp(line, "quit") == 0) {
                                toQuit = 1;     // kilépés jelzése
                        }
                }
                free(line);                     // felszabadítás
        }
        write_history("./trl.history");
        return 0;
}
```

A program a bemenetről beolvasott sorokat megjeleníti. Ha a bemeneten a quit szöveget gépeljük be, vagy a Ctrl-D billentyű-kombinációt ütjük le, akkor kilép. A begépelt sorokat egy trl.history nevű fájlban tárolja el, így későbbi futtatásokkor is visszaléphetünk a korábban begépelt sorokra (lásd még 24.).

[8] A 4.0-ás változatot az ANSI C header fájlok miatt választottuk, bár minden elem a 2.1-es változatban is megvan

[9] Itt a "~" jel a HOME könyvtárra utal.

Java megoldás

Induljunk el felülről lefelé a natív osztályok megírásakor. A C nyelvű programnak megfelelő Java alkalmazás a következő:

```
import java.io.IOException;

/** Példa a History és Readline osztályok használatára. */
public class JRL {
    public static void main(String args[]) {
        String      line = null;
        boolean     toQuit = false;

        // attribútumok lekérdezése
        System.out.println(Readline.getName());
        System.out.println(Readline.getVersion());

        // korábban bevitt sorok beolvasása
        try {
            History.read("jrl.history");
        } catch(IOException e) {
            System.err.println("jrl.history olvasása sikertelen");
        }

        while(! toQuit) {
            // egy sor beolvasása
            line = Readline.readline("ez itt a prompt helye> ");

            // Ctrl-D vagy fájlvége jel
            if(line == null) break;

            if(line.length() > 0) {
                System.out.println(line);
                History.add(line);
                toQuit = (line.compareTo("quit") == 0);
            }
        }

        // az új sorok kiírása
        try {
            History.write("jrl.history");
        } catch(IOException e) {
            System.err.println("jrl.history írása sikertelen: "
                + e.getMessage());
        }
    }
}
```

Mint látható, a történeti tömbbel és a beolvasással kapcsolatos funkciók két osztályba vannak szétválasztva: *History* és *Readline*.

Szövegkonverzió

Magát a beolvasást végző readline metódus ismertetése a példában előforduló problémák nagy részét lefedi, ezért csak ezzel foglalkozunk részletesebben. A teljes forráskód a hozzá tartozó segédfájlokkal együtt a CD-n található meg.

A *Readline* osztályban a metódus deklarációja a következő:

```
public static native String readline(String prompt);
```

Ebből a következő C nyelvű deklaráció generálódik:

```
JNIEXPORT jstring JNICALL Java_Readline_readline
    (JNIEnv *, jclass, jstring);
```

A függvényt a következő — általunk megírt — törzs valósítja meg:

```
JNIEXPORT jstring JNICALL
Java_Readline_readline (JNIEnv *env, jclass clazz, jstring prompt) {
    char*       line;
    const char* ccprompt;
    char*       cprompt;

    if(prompt == NULL) {              // üres-e a prompt
        cprompt = NULL;
    } else {                          // C szöveggé alakítás
        ccprompt = (*env)->GetStringUTFChars(env, str, JNI_FALSE);
        cprompt = strdup(ccprompt);
        (*env)->ReleaseStringUTFChars(env, prompt, ccprompt);
    }
    line = readline(cprompt);         // a lényeg
    if(cprompt != NULL) free(cprompt);

    if(line == (char*)NULL) {         // üres-e a sor
        return NULL;
    } else {                          // Java szöveggé alakítás
        return (*env)->NewStringUTF(env, line);
        free(line);
    }
}
```

A megvalósítás kényes pontja a szövegek konvertálása.

A *C szöveggé alakítás* során először a GetStringUTFChars függvénnyel nyerjük ki a Java szöveget karakterenkénti 8 bites ábrázolásban. Ez a módszer az ASCII karakterkészleten gond nélkül működik, de azon kívül további konverziót igényelne a helyi karakterkészletnek megfelelően. Az ékezetes karaktereken ezért meglepő eredményt produkálhat.

Az így kapott szöveget csak konstans függvénynek lehet paraméterül adni, sajnos azonban a C nyelven megírt forrásokban a függvénydeklarációk nagy része még akkor sem tartalmazza a const módosító szót, ha a paraméter értékét nem változtatja meg. Ha tudjuk, hogy mit csinálunk, akkor típuskényszerítéssel is kísérletezhetnénk, a korrekt megoldás azonban a szöveg lemásolása: strdup.

A másolat elkészítése után a konstans szövegre már nincs szükségünk, így azt a ReleaseStringUTFChars függvénnyel fel is szabadíthatjuk.

A *Java szöveggé alakítás* kevésbé bonyolult, ugyanis a NewStringUTF függvény meghívásával elvégezhető. Az ASCII karakterkészleten felüli karakterek konverziója itt is problémás.

A konvertálások során a C nyelvű kódban dinamikusan lefoglalt memóriaterületeket ne felejtsük el felszabadítani!

Kivételek kezelése

A natív programrészekben is előfordulhatnak kivételes helyzetek vagy hibák, melyeket a Java kódhoz kivételek formájában illik eljuttatni.

A read_history függvény nullától eltérő visszatérési értékét is kivételen keresztül kell visszaadni. A *History* osztály megfelelő metódusának deklarációja a következő:
`public static native void read(String filename) throws IOException;`

A C nyelven megírt törzsben a következő programrészlettel válthatjuk ki a megjelölt (IOException) kivételt:

```
retval = read_history(cfilename);
...
if(retval != 0) {
    cIOException = (*env)->FindClass(env, "java/io/IOException");
    (*env)->ThrowNew(env, cIOException, strerror(retval));
}
```

A kivételeket a *JRL* alkalmazásban a „jrl.history” fájl törlésével (`rm jrl.history`; olvasási hiba), vagy csak olvasható állapotba hozásával (`chmod -w jrl.history`; írási hiba) tesztelhetjük.

Regisztrálás

A dinamikus kötés „kézi vezérléssel” is történhet (lásd 27.3.3.). Ilyenkor a behívott osztott könyvtár függvényeinek Java metódusokhoz rendelése nem a beépített név szerinti megfeleltetés, hanem a programozó szándéka szerint történik. Ennek szemléltetésére a *History* osztályban hozzuk létre a registerNatives metódust, melyet a betöltés után (loadLibrary) rögtön meg kell hívni [10]:
`private static native void registerNatives();`

A metódus megvalósításában a metódusok nevéhez és szignatúrájához egyértelműen hozzárendeljük a megfelelő C függvényeket:

```
JNINativeMethod     methods[] = {
    {"read", "(Ljava/lang/String;)V", hist_read},
    {"write", "(Ljava/lang/String;)V", hist_write},
    {"add", "(Ljava/lang/String;)V", hist_add}
};

JNIEXPORT void JNICALL
Java_History_registerNatives (JNIEnv *env, jclass clazz) {
    (*env)->RegisterNatives(env, clazz, methods,
            sizeof(methods)/sizeof(JNINativeMethod));
}
```

A kötések a következő táblázatnak felelnek meg:

Java metódusnév	alapértelmezés	jelenlegi
read	Java_History_read	hist_read
write	Java_History_write	hist_write
add	Java_History_add	hist_add

Az alkalmazás „bőbeszédű” (-verbose:jni) futtatása során nagyjából a következő sorokat láthatjuk a képernyőn:

[10] A JNI_OnLoad függvény nem alkalmas erre, mert nincs meg benne az összes adat: szál leírója (env), osztály hivatkozása (clazz).

```
java-readline> java -verbose:jni JRL
[Dynamic-linking native method java/lang/Object.registerNatives ... JNI]
[Registering JNI native method java/lang/Object.hashCode]
...
[Loaded native library: /home/szamcsi/java-readline/libReadline.so]
[Dynamic-linking native method Readline.setName ... JNI]
[Loaded native library: /home/szamcsi/java-readline/libHistory.so]
[Dynamic-linking native method History.registerNatives ... JNI]
[Registering JNI native method History.read]
[Registering JNI native method History.write]
[Registering JNI native method History.add]
```

Mint látható magának a registerNatives metódusnak a hozzárendelését még a beépített automatikus algoritmus végzi, azonban a további metódusokat már a fenti programrészlet kapcsolja a Java metódustábla megfelelő pontjaihoz.

Fordítás

A fordításhoz és futtatáshoz alapvetően a *readline* csomagra van szükség. Ha ez nem áll rendelkezésre, akkor a readline-4.0.tar.gz állományt letöltve (CD-n is megtalálható) magunk is telepíthetjük:

```
gunzip readline-4.0.tar.gz
tar -xvf readline-4.0.tar
cd readline-4.0
./configure
make
make shared
make install
make install-shared
```

Ha a *curses*, vagy *ncurses* csomag is telepítve van a gépen, akkor érdemes a
`./configure --with-curses`
paranccsal konfigurálni a fordítás előtt. Ez azzal jár, hogy a natív kódok linkelési fázisában ezt a könyvárat is meg kell adni (-lcurses).

A natív kódrészletek fordítását megkönnyítendő Makefile-okat mellékelünk. Ezek GNU fejlesztő eszközök meglétét feltételezik: GNU C fordítót és GNU make programot.

A fordítás kétféleképpen történhet:

1. A forrásokat tartalmazó könyvárban a make parancs kiadásával.
2. Architektúrától függő alkönyvtárban az archmake parancs hatására (pl. ugyanazt a könyvtárat Linux és Solaris alatt is látjuk).

Az első esetben a bináris állományok a források könyvtárában, a második esetben egy alkönyvtárban generálódnak. A Java bájtkódokat tartalmazó .class fájlok mindkét esetben a források mellé generálódnak, hiszen hordozhatóak.

A make, illetve az archmake parancs lehetséges paraméterei:

trl elkészíti a trl programot.
(semmi) lefordítja a futtatáshoz szükséges állományokat.
run lefuttatja a JRL Java alkalmazást.
clean letörli a fordítás közben keletkezett mellékes fájlokat.
purge minden generált fájlt letöröl.

A fordítás helyi jellemzőit a Makefile.OS nevű állományban lehet beállítani (az ‘uname -s‘ parancs eredménye az OS végződés):

JDK_ROOT a fejlesztő-környezet helye (amelyben az include alkönyvtár található).
JDK_ARCH az architektúra neve; a $JDK_ROOT/include/ könyvtárban lévő alkönyvtár neve (pl.: solaris vagy linux).
EXTRA_LIBS a linkeléshez szükséges könyvtárak (pl. -lcurses).

A közös szabályok a Makefile.common állományban vannak.
Példaként álljon itt a *Linux* alkönyvtárban végzett fordítás:

```
$ make
gcc  -c -fPIC  -I. -I/usr/local/jdk1.2/include -I/usr/local/jdk1.2/include
/linux -Wall -O2 -o History.o ../History.c
gcc  -c -fPIC  -I. -I/usr/local/jdk1.2/include -I/usr/local/jdk1.2/include
/linux -Wall -O2 -o jni_util.o ../jni_util.c
gcc  -shared -o libHistory.so History.o jni_util.o -lreadline  -lcurses
javah -jni -classpath .. Readline
gcc  -c -fPIC  -I. -I/usr/local/jdk1.2/include -I/usr/local/jdk1.2/include
/linux -Wall -O2 -o Readline.o ../Readline.c
gcc  -shared -o libReadline.so Readline.o jni_util.o -lreadline  -lcurses
$ make run
LD_LIBRARY_PATH=:. java -classpath .. JRL
```