Q. Fejezet

JVM: Java Virtuális Gép

Ez a fejezet a Java Virtuális Gép (JVM, Java Virtual Machine) rövid leírását tartalmazza. Célja, hogy áttekintést adjon a JVM-ről nemcsak azoknak, akik szeretnének egy új implementációt készíteni, hanem a JVM kódot előállító fordítóprogramok íróinak is. Szeretnénk bemutatni, hogy a JVM modern, egyszerű és hatékony virtuális gép, amely igen jól használható hordozható és/vagy hálózati alkalmazások készítésére.

A fejezet további részeiben a JVM általános jellemzése után bemutatjuk egy tipikus JVM gép felépítését, a .class fájlok formátumát és a JVM gépi kód utasításkészletét.

Q.1. A JVM jellemzői

A JVM egy képzeletbeli számítógép, amelyet szoftveresen „szimulálunk" valamilyen adott konkrét számítógépen. A JVM .class fájlokból olvassa be a végrehajtandó kódot, indításakor a megadottban, public módosítóval ellátott osztály kódját kezdi végrehajtani. Természetesen ez az osztály meghívhatja más osztályok metódusait is. A JVM specifikációja pontosan meghatározza az elfogadható .class fájlok felépítését, és megadja, hogyan kell a szabványos Java könyvtárakat JVM-re megvalósítani. Ez utóbbi témakört (terjedelmi okokból) nem tárgyaljuk.

Ugyanaz a Java forrásprogram bárhol, bármilyen fordítóval lefordítva bájtról-bájtra ugyanazt a kódot adja, ennek végrehajtásakor pedig ugyanazt az eredményt kapjuk minden környezetben. Ez – a lehetőségekhez képest – még a programok megjelenésére (felépítésére) is igaz, ezt szolgálják a szabványosított könyvtárak, és azok pontosan meghatározott megvalósítása a JVM-en. Lesznek elkerülhetetlen különbségek a megjelenítésben, például a grafikus ablakozó-rendszerek eltérő sajátosságai miatt, de ezt a nyelv tervezői igyekeznek minimális szintre szorítani.

A program gépi kódja és eredménye tehát csak a forrásprogramtól függ. Érdemes megemlíteni, hogy a JVM által végrehajtható kódot nem csak Java nyelvű forrásprogramból lehet előállítani, léteznek fordítók más nyelvekhez (például Adához) is.

A közös gépi kód előnye a hordozhatóság mellett abban van igazán, hogy a már lefordított (így igen tömör) program kis mérete miatt hatékonyan továbbítható hálózaton keresztül. Érthető, hogy a Java és a JVM megjelenését az Internet, és ezen belül a WWW megjelenése gyorsította. Maga a virtuális gép mint a fordítók célgépe nem új koncepció. Találkozhattunk vele már például az IBM vagy a Digital nagyszámítógépek mikrokódjánál, az UCSD Pascal nyelv p-kódjánál, vagy – az előbbiektől kicsit eltérő köntösben – a MACH mikrokernel kapcsán.

A JVM-et igyekeztek úgy megtervezni, hogy minél többféle környezetben hatékonyan futtatható legyen, és könnyű legyen rajta futtatható kódot előállító fordítóprogramokat készíteni. Ezek a követelmények magukkal hozták a JVM hátrányait is: ha a JVM tervezői által felállított minimális követelményeket nem teljesíti az adott architektúra (pl. a processzor nem tud 32 bites egészekkel számolni, vagy nincs lebegőpontos koprocesszor), akkor készíthetünk ugyan hozzá JVM-futtató programot, de ez nagyon lassú, és készítése nagyon nehézkes lesz. További hátrány, hogy sok, a mai gépekre jellemző, de a jövőben várhatóan meghaladott korlátot eleve beépítettek. Például a JVM 32 bites címtartományt feltételez, ami 64 bites címzésű gépeken külön „trükköket" igényel a megvalósításakor. A másik irányban még nehezebb a JVM-készítő programozó dolga: képzeljük el, hogyan

írnánk meg a JVM-et egy 16 bites címzésű processzorra. A helyzet nem reménytelen, a legtöbb mai elterjedt processzorfajtán a JVM könnyen és hatékonyan megvalósítható.

Interpreteren kívül készíthető a JVM-hez JVM-futtató célhardver, amelynek JVM a gépi kódja, és készíthetünk olyan fordítóprogramot is, amelyik a JVM kódot valamely gép saját gépi kódjára fordítja. A JVM legfontosabb felhasználási területe azonban ma még a különféle hálózati alkalmazásokban van: a JVM egyedülállóan elegáns megoldást kínál programok hálózaton keresztül történő terjesztésére.

Hatékonyságát pedig nagymértékben javítja, hogy lehet olyan metódusokat is írni, amelyeket nem a JVM hajt végre, hanem C vagy más nyelven megírt, és az adott célgép gépi kódjára lefordított függvényekkel valósítunk meg. Például a standard könyvtáraknak az operációs rendszerrel kapcsolatot tartó részei is ilyen módon működnek.

Q.2. A JVM felépítése

A JVM 8, 16, 32 és 64 bites egészekkel, egyszeres és duplapontosságú lebegőpontos számokkal, Unicode szabvány szerinti karakterekkel, valamint 32 bites memóriacímekkel dolgozik. A számításokhoz regiszterek helyett vermet használ, a veremben és a memóriában a konstansokat és változókat 32 bites egységekben (ún. szavakban) tárolja. Párhuzamos, többprocesszoros környezetben az egyes objektumok szinkronizált eléréséhez egy monitorszerű mechanizmust alkalmaz. Beépített lehetőségei vannak hiba- és kivételkezelésre, a programfejlesztés támogatására, például nyomkövetésre, hibakeresésre.

A mai modern processzorok általában teljesítik ezeket a követelményeket, így a JVM hatékonyan megvalósítható rajtuk. Szintén megoldható az is, hogy a JVM 32 bites címzése helyett 64 bites címtartományú processzort használjunk.

A JVM-ben a következő adattípusok léteznek:

- byte: 8 bites előjeles egész szám.
- short: 16 bites előjeles egész szám.
- int: 32 bites előjeles egész szám.
- long: 64 bites előjeles egész szám.
- float: 32 bites egyszeres pontosságú lebegőpontos szám.
- double: 64 bites kétszeres pontosságú lebegőpontos szám.
- char: 16 bites Unicode szabvány szerinti karakter.

A Virtuális Gép szintjén nincs külön boolean típus, logikai értékek esetén az int típus típusműveleteivel kell számolni, a boolean tömbök pedig byte tömbként tárolódnak. További adattípus a returnAddress, amely 32 bites tárcímet tárolnak, egy objektum címét, vagy a jsr/ret/jsr_w utasítások visszatérési címét tartalmazza.

A JVM egyik fontos követelménye, hogy az egyes gépi utasításokat mindig a megfelelő típusú operandusokkal kell használni. Tilos például betenni a verembe két int értéket, majd long-ként felhasználni azokat. A JVM hibásnak tekinthet minden olyan JVM kódot, amely nem tartja be ezt az elvet.

Végrehajtás előtt a betöltött .class fájlra számos egyéb ellenőrzést is tesz a JVM, például ellenőrzi, hogy egy interfészt megvalósító osztály az interfészben szereplő valamennyi metódust implementálta-e.[1]

Tipikus JVM megvalósítások a memória nagy részét egyetlen garbage collected heap[2]-ként kezelik, ebben tárolják az egyes objektumokat, innen foglalják le a szükséges memóriaterületeket. A .class fájlok metódusai is a heapből foglalnak helyet, de kerülhet a metódusok kódja a heapen kívüli, elkülönített memóriaterületre is.

[1] Ez az ellenőrzés csak a JVM 1.1.x verzióktól szerepel.

[2] A heap egy, a program statikus struktúrájától függetlenül kezelt memóriaterület. A garbage collected jelző pedig arra utal, hogy a heap memóriaterülettel való gazdálkodást egy szemétgyűjtő algoritmus irányítja.

A továbbiakban a metódusokhoz tartozó memória felépítését vizsgáljuk.
A JVM minden pillanatban egyetlen metódus kódját hajtja végre. A következő végrehajtandó utasítás címét a PC regiszter tartalmazza. Ezen kívül minden metódushoz tartozik még három memóriaterület, ezek címe a vars, optop és frame regiszterekben található.

- Változók

  A vars regiszterrel megadott címen találjuk az adott metódus lokális változóit. Minden változóra a vars regiszterhez viszonyított eltolás (offset) megadásával hivatkozhatunk. A változók majdnem mind 32 bitesek, kivétel a long és double típusú változók, amelyek két egymás utáni helyet foglalnak el. Nincs megkötve, hogy ebben a két szóban milyen sorrendben tároljuk a long változó magas és alacsony helyiértékű bájtjait, illetve hogy hogyan ábrázoljuk a double típusú értékeket.
- Operandusverem

  A legtöbb JVM gépi utasítás az operandusvermet használja a művelet elvégzésére. Az operandusverem kezdőcímét az optop regiszter tartalmazza. Külön utasítások szolgálnak az operandusverem és a lokális változók közötti adatmozgatásra, illetve az egyes műveletek elvégzésére az operandusverem segítségével: például egy összeadás esetén be kell tennünk a verembe az összeadandókat, majd az összeadó utasítás a verem tetején levő két számot leveszi a veremről, és az eredményt a verem tetejére teszi.

  A verem használatával kiküszöbölhető az a probléma, ami abból származik, hogy egyes processzorokon túl kevés regiszter található; a verem egyformán hatékonyan megvalósítható valamennyi processzorfajtán.

  Az operandusverem a lokális változókhoz hasonlóan mindig 32 bites szavakat tárol, a byte vagy short típusok is egy egész szót elfoglalnak, a long és double típusok pedig két egymás utáni helyet használnak fel a veremben.
- Végrehajtási környezet

  Minden metódushoz tartozik egy végrehajtási környezet, amelynek kezdetére a frame regiszter mutat. A végrehajtási környezetben tárolhatunk minden további információt, ami a futtatáshoz szükséges. Ilyenek lehetnek például a szimbolikus metódushívások feloldásához (dynamic linking) szükséges szimbólumtáblák; egy metódus befejezésekor a hívó metódus regisztereinek visszatöltéséhez, vagy a hibakezeléshez szükséges adatok. De tárolhatunk itt további implementációfüggő adatokat, például nyomkövetéshez szükséges táblázatokat is.

Q.3. Egy .class fájl formátuma

A JVM bemenetéül szolgáló .class fájlokban egy Java osztály vagy egy Java interfész lefordított változata található. Minden JVM implementációnak képesnek kell lennie az összes szabályos .class fájl kezelésére.

A .class fájlok 8 bites bájtokból állnak. Az ennél hosszabb, 16 vagy 32 bites mennyiségeket egymás utáni bájtok tartalmazzák. A bájtok sorrendje kötött: mindig a magasabb helyiértékű bájt következik először.

A .class fájlok formátuma a következő struktúrával írható le (u1, u2, u4 előjel nélküli 1, 2 és 4-bájtos számokat jelölnek):

```
ClassFile {
        u4 magic;
        u2 minor_version;
        u2 major_version;
```

```
        u2 constant_pool_count;
        cp_info constant_pool[constant_pool_count - 1];
        u2 access_flags;
        u2 this_class;
        u2 super_class;
        u2 interfaces_count;
        u2 interfaces[interfaces_count];
        u2 fields_count;
        field_info fields[fields_count];
        u2 methods_count;
        method_info methods[methods_count];
        u2 attributes_count;
        attribute_info attributes[attributes_count];
}
```

Az egyes mezők jelentését egy példa segítségével mutatjuk be bemutatni. Adott egy .class fájl, próbáljuk meg a fájl listája alapján megállapítani, hogy milyen forrásprogramból készíthette a fordító.

A .class fájlról az od (oktális lista, octal dump) UNIX paranccsal készítettünk egy listát, melyet megjegyzésekkel láttuk el. A listában a hexadecimális címek után az első sorban hexadecimálisan, majd a következő sorban karakteresen láthatjuk a fájl tartalmát, 16 bájtot soronként. A hexa értékek fölé a jelentésre utaló megjegyzéseket írtunk, a kód részeit függőleges vonalkákkal választottuk el egymástól.

A tárgyalásban, ahol külön másként nem jelezzük, hexadecimális számokat fogunk használni.

```
          magic      verzio     cpc   1        2
000000 ca fe ba be|00 03 00 2d|00 20|08|00 13|07|00 14|
       312 376 272 276  \0 003  \0   -  \0      \b  \0 023  \a  \0 024

        3        4        5        6             7
000010 07|00 1a|07|00 1b|07|00 1c|0a|00 04|00 09|09|00
        \a  \0 032  \a  \0 033  \a  \0 034  \n  \0 004  \0  \t  \t  \0

                8             9             a
000020 05|00 0a|0a|00 03|00 0b|0c|00 0f|00 0c|0c|00 1e|
       005  \0  \n  \n  \0 003  \0  \v  \f  \0 017  \0  \f  \f  \0 036

              b             c                d
000030 00 17|0c|00 1f|00 0d|01|00 03|28 29 56|01|00 15|
        \0 027  \f  \0 037  \0  \r 001  \0 003   (   )   V 001  \0 025


000040 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69
        (   L   j   a   v   a   /   l   a   n   g   /   S   t   r   i

                      e
000050 6e 67 3b 29 56|01|00 16|28 5b 4c 6a 61 76 61 2f
        n   g   ;   )   V 001  \0 026   (   [   L   j   a   v   a   /


                                                f
000060 6c 61 6e 67 2f 53 74 72 69 6e 67 3b 29 56|01|00
        l   a   n   g   /   S   t   r   i   n   g   ;   )   V 001  \0
```

```
                              10                      11
000070 06|3c 69 6e 69 74 3e|01|00 04|43 6f 64 65|01|00
       006   <  i  n  i  t  > 001  \0 004   C  o  d  e 001  \0

                                                      12
000080 0d|43 6f 6e 73 74 61 6e 74 56 61 6c 75 65|01|00
       \r  C  o  n  s  t  a  n  t  V  a  l  u  e 001  \0

                                          13
000090 0a|45 78 63 65 70 74 69 6f 6e 73|01|00 0c|48 65
       \n  E  x  c  e  p  t  i  o  n  s 001  \0  \f  H  e

                                       14
0000a0 6c 6c 6f 20 76 69 6c 61 67 21|01|00 0a|48 65 6c
        l  l  o     v  i  l  a  g  ! 001  \0  \n  H  e  l

                              15
0000b0 6c 6f 56 69 6c 61 67|01|00 0f|48 65 6c 6c 6f 56
        l  o  V  i  l  a  g 001  \0 017  H  e  l  l  o  V

                                    16
0000c0 69 6c 61 67 2e 6a 61 76 61|01|00 0f|4c 69 6e 65
        i  l  a  g  .  j  a  v  a 001  \0 017  L  i  n  e

                                          17
0000d0 4e 75 6d 62 65 72 54 61 62 6c 65|01|00 15|4c 6a
        N  u  m  b  e  r  T  a  b  l  e 001  \0 025  L  j

0000e0 61 76 61 2f 69 6f 2f 50 72 69 6e 74 53 74 72 65
        a  v  a  /  i  o  /  P  r  i  n  t  S  t  r  e

                 18
0000f0 61 6d 3b|01|00 0e|4c 6f 63 61 6c 56 61 72 69 61
        a  m  ; 001  \0 016  L  o  c  a  l  V  a  r  i  a

                    19
000100 62 6c 65 73|01|00 0a|53 6f 75 72 63 65 46 69 6c
        b  l  e  s 001  \0  \n  S  o  u  r  c  e  F  i  l

          1a
000110 65|01|00 13|6a 61 76 61 2f 69 6f 2f 50 72 69 6e
        e 001  \0 023  j  a  v  a  /  i  o  /  P  r  i  n

                                1b
000120 74 53 74 72 65 61 6d|01|00 10|6a 61 76 61 2f 6c
        t  S  t  r  e  a  m 001  \0 020  j  a  v  a  /  l

                                          1c
000130 61 6e 67 2f 4f 62 6a 65 63 74|01|00 10|6a 61 76
        a  n  g  /  O  b  j  e  c  t 001  \0 020  j  a  v

                                                1d
000140 61 2f 6c 61 6e 67 2f 53 79 73 74 65 6d|01|00 04|
        a  /  l  a  n  g  /  S  y  s  t  e  m 001  \0 004
```

```
                  1e                1f
000150 6d 61 69 6e|01|00 03|6f 75 74|01|00 07|70 72 69
        m   a   i   n 001  \0 003   o   u   t 001  \0  \a   p   r   i

                 access this super interf field methods
000160 6e 74 6c 6e|00 21|00 02|00 04|00 00|00 00|00 02|
        n   t   l   n  \0   !  \0 002  \0 004  \0  \0  \0  \0  \0 002

       access name  sign attrib name    length   stack
000170 00 09|00 1d|00 0e|00 01|00 10|00 00 00 25|00 02|
        \0  \t  \0 035  \0 016  \0 001  \0 020  \0  \0  \0   %  \0 002

       locals codelength|getstat07 ldc1 invvirt ret|exc. table length
000180 00 01|00 00 00 09|b2 00 07 12 01 b6 00 08 b1|00
        \0 001  \0  \0  \0  \t 262  \0  \a 022 001 266  \0  \b 261  \0

          attrib name   length
000190 00|00 01|00 16|00 00 00 0a|00 02 00 00 00 03 00
        \0  \0 001  \0 026  \0  \0  \0  \n  \0 002  \0  \0  \0 003  \0

               |access this super interf
0001a0 08 00 02|00 01|00 0f|00 0c|00 01|00 10 00 00 00
        \b  \0 002  \0 001  \0 017  \0  \f  \0 001  \0 020  \0  \0  \0


0001b0 1d 00 01 00 01 00 00 00 05 2a b7 00 06 b1 00 00
       035  \0 001  \0 001  \0  \0  \0 005   * 267  \0 006 261  \0  \0


0001c0 00 01 00 16 00 00 00 06 00 01 00 00 00 01 00 01
        \0 001  \0 026  \0  \0  \0 006  \0 001  \0  \0  \0 001  \0 001


0001d0 00 19 00 00 00 02 00 15
        \0 031  \0  \0  \0 002  \0 025


0001d8
```

Kezdjük el olvasni a fájlt az elejétől, és illesszük a ClassFile fentebb bemutatott felépítéséhez! Az olvasónak javasoljuk, hogy próbálja a listában követni a leírtakat.

```
          magic      verzio     cpc   1        2
000000 ca fe ba be|00 03 00 2d|00 20|08|00 13|07|00 14|
       312 376 272 276  \0 003  \0  -  \0    \b  \0 023  \a  \0 024
```

magic

A fájl első négy bájtja 0xCAFEBABE; minden szabályos .class fájlnak így kell kezdődnie. Innen állapítható meg, ha véletlenül nem .class fájlt akarunk végrehajtani.

verziószám

A következő 2-2 bájt a .class fájlt előállító fordítóprogram al- és főverziószáma (minor, major verziószám). Ezeket F.A alakban írjuk, ahol F a fő-, A az alverziószám. A verziószámok rendezése lexikografikusan történik, és megállapításuk a Sun kizárólagos joga.

Példánkban a verziószám 45.3.

constant_pool_count

A következő két bájt a konstansterület (constant_pool) elemeinek számát tartalmazza, jelen esetben ez 0x20, azaz 32. A konstansterület 0. elemét a JVM nem használja, így a fájlban ezután a konstansterület 31 darab bejegyzése következik.

constant_pool

A konstansterület (constant_pool) bejegyzései változó hosszúságúak, az egyes bejegyzések első bájtja határozza meg, hogy az adott bejegyzést hogyan kell értelmezni. A következő táblázat mutatja ezen első bájt lehetséges értékeit:

Konstans típusa	értéke
CONSTANT_Utf8	1
CONSTANT_Unicode	2
CONSTANT_Integer	3
CONSTANT_Float	4
CONSTANT_Long	5
CONSTANT_Double	6
CONSTANT_Class	7
CONSTANT_String	8
CONSTANT_Fieldref	9
CONSTANT_Methodref	A
CONSTANT_InterfaceMethodRef	B
CONSTANT_NameAndType	C

A mi példánkban a következő bájt értéke 8.

String

Az előző táblázat szerint a 8-as érték egy Stringet jelöl, amely a következő két bájtban egy indexet tartalmaz:

```
CONSTANT_String_info {
        u1 tag; //=CONSTANT_String
        u2 sztring_index;
}
```

A tag mező értéke tehát most 8, a következő két bájt 0x0013, ami azt jelenti, hogy a konstansterület első bejegyzése constant_pool[1] egy String, amelynek értékét (magát a szöveget) a konstans terület 0x13. eleme fogja tartalmazni Utf8 kódolással. (Az Utf8 kódolást is hamarosan ismertetjük.)

```
         magic      verzio     cpc   1        2
000000 ca fe ba be|00 03 00 2d|00 20|08|00 13|07|00 14|
       312 376 272 276  \0 003  \0   -  \0      \b  \0 023  \a  \0 024
```

(A hexadecimális sor feletti számok jelölik a konstansterület elemeinek első bájtját és a konstansterületbeli sorszámát.)

class

A listát tovább olvasva 0x07-es érték következik, vagyis a konstansterület második eleme egy Class. Ennek felépítése hasonló az előzőhöz, vagyis a 07 bájt után két bájton újra egy konstansterületbeli index következik, ahol ennek az osztálynak a nevét fogjuk találni.

Példánkban ez 0x0014, vagyis a konstansterület 0x14. helyén fogjuk az osztály nevét megtalálni.

A lista első sorával ezzel végeztünk, a következő sor elején még három Class található, amelyek neve a konstansterület 0x001A, 0x001B és 0x001C. bejegyzéseiben található.

```
       3       4       5       6              7
000010 07|00 1a|07|00 1b|07|00 1c|0a|00 04|00 09|09|00
       \a  \0 032  \a  \0 033  \a  \0 034  \n  \0 004  \0  \t  \t  \0

              8             9             a
000020 05|00 0a|0a|00 03|00 0b|0c|00 0f|00 0c|0c|00 1e|
       005  \0  \n  \n  \0 003  \0  \v  \f  \0 017  \0  \f  \f  \0 036
```

MethodRef

A konstansterület következő (6.) eleme a 0x0A számmal kezdődik, a fent bemutatott táblázat szerint itt egy Methodref (metódus-leíró referencia) található. Ennek felépítése a következő:

```
CONSTANT_Methodref_info {
        u1 tag; //=CONSTANT_Methodref
        u2 class_index;
        u2 name_and_type_index;
       }
```

A tag komponensének az értéke 0x0A, class_index (értéke most 4) a metódust tartalmazó osztályt adja meg; amint az előzőekben már megállapítottuk, a konstansterület negyedik eleme egy Class. A következő két bájt (0x0009) szintén a konstansterület indexe lesz, itt fogjuk megtalálni a metódus nevét és paramétereinek ill. visszatérési értékének típusát.

Field

A következő (7.) bejegyzés egy mező. Ennek felépítése teljesen ugyanaz, mint a metódusok esetén, csak itt class_index a mezőt tartalmazó osztályt adja meg, name_and_type_index pedig a mező típusát írja le. Jelen esetben a tartalmazó osztály az 5. bejegyzésben, a típus a 0x0a. bejegyzésben található.

Ezután ismét egy metódus (8.) következik a listában (közben átléptünk a harmadik sorba), ez a metódus a constant_pool[3] osztályba tartozik, neve és paramétereinek típusa a konstansterület 0x0b. elemében található.

Foglaljuk össze, mit tartalmaz eddig a .class fájl, mit tudtunk meg idáig:

- helyes mágikus számmal kezdődik (0xCAFEBABE), vagyis ez vélhetőleg egy helyes .class fájl lesz,
- a fordító verziója 45.3,
- a konstansterület 31 elemet fog tartalmazni,
- ismerjük idáig a konstansterület első hét elemét:
  1. az első elem egy String, amelynek szövege a 0x13. bejegyzésben található,
  2. a második elem egy osztály, melynek neve a konstansterület 0x14. eleme (constant_pool[0x14]),

3. ismét egy osztály, neve constant_pool[0x1A]-ban van,
4. osztály, neve constant_pool[1B],
5. osztály, neve constant_pool[1C],
6. ez egy metódus, az őt tartalmazó osztály a constant_pool[4], nevét és típusát a constant_pool[9] tartalmazza,
7. mező, melyet a constant_pool[5]-ben található nevű osztály tartalmaz, típusa a 0x0a. bejegyzésben van leírva,
8. metódus, a tartalmazó osztály a 3., a metódus neve és típusa a 0x0b. bejegyzésben van.

```
        3        4        5        6              7
000010 07|00 1a|07|00 1b|07|00 1c|0a|00 04|00 09|09|00
        \a  \0 032  \a  \0 033  \a  \0 034  \n  \0 004  \0  \t  \t  \0

                 8             9             a
000020 05|00 0a|0a|00 03|00 0b|0c|00 0f|00 0c|0c|00 1e|
       005  \0  \n  \n  \0 003  \0  \v  \f  \0 017  \0  \f  \f  \0 036
```

Name_And_Type

A konstansterület 9. bejegyzése a 0x0C értékkel kezdődik, tehát itt egy név és típus megadására alkalmas Name_And_Type bejegyzés található:

```
CONSTANT_NameAndType_info {
        u1 tag; //=CONSTANT_NameAndType
        u2 name_index;
        u2 signature_index;
        }
```

A name_index egy konstansterületbeli index, itt találjuk a kérdéses metódus, mező vagy tömb nevét. signature_index pedig szintén egy a konstansterületre mutató index, itt lesz a metódus (vagy mező vagy tömb) „aláírása".

Signature

Egy metódus, mező vagy tömb típusát leíró sztringet aláírásnak (signature) nevezünk. Az aláírást a következő szabályok szerint képezzük:

- az alaptípusok aláírása egy betű: B byte, C char, D double, F float, I int, J long, S short és Z boolean érték esetén,
- egy adott osztályba tartozó objektum jelölése: L<osztálynév>;

  ahol <osztálynév> az osztály teljes minősített neve. Például az Object osztály aláírása: Ljava/lang/Object;,
- tömbök esetén egy adott elemtípushoz és pozitív egész számmal megadott (opcionális) mérethez a következő a jelölés: [<méret><elemtípus>,
- metódusok esetén az argumentumok aláírását közvetlenül egymás után írva kerek zárójelbe tesszük, majd közvetlen ezután megadjuk a metódus visszatérési értékének típusát vagy V-t, ha nem ad vissza értéket a metódus (void). Például egy paraméter nélküli, void metódus aláírása: ()V

A konstans lista 0x09., 0.0a. és 0x0b. elemei Name_And_Type bejegyzések.

```
                8             9             a
000020 05|00 0a|0a|00 03|00 0b|0c|00 0f|00 0c|0c|00 1e|
       005  \0  \n  \n  \0 003  \0  \v  \f  \0 017  \0  \f  \f  \0 036

              b             c             d
000030 00 17|0c|00 1f|00 0d|01|00 03|28 29 56|01|00 15|
        \0 027  \f  \0 037  \0  \r 001  \0 003   (   )   V 001  \0 025
```

Utf8

A .class fájl listáját tovább olvasva a következő (0x0c.) bejegyzés a 0x01 bájttal kezdődik, vagyis ez egy Utf8 sztring lesz:

```
CONSTANT_Utf8_info {
        u1 tag; //=CONSTANT_Utf8
        u2 length;
        u1 bytes[length];
        }
```

A 01 bájt után találjuk a sztring hosszát bájtokban, majd a sztring bájtjait Utf8 kódolás szerint.

Az Utf8 kódolás 1 és 127 közötti kódú karakterek esetén egyetlen bájtot használ: a karakter Utf8 kódja megegyezik a karakter ASCII kódjával. A 0 kódú karakter és a Unicode szabvány szerinti további (két bájtban leírható) nemzetközi karakterek 2 vagy 3 bájtban kódolva szerepelnek.

Példánknál maradva, a konstansterület 0x0c. eleme egy 3 karakter hosszú szöveg, tartalma: "()V"

```
              b             c             d
000030 00 17|0c|00 1f|00 0d|01|00 03|28 29 56|01|00 15|
        \0 027  \f  \0 037  \0  \r 001  \0 003   (   )   V   001  \0 025


000040 28 4c 6a 61 76 61 2f 6c 61 6e 67 2f 53 74 72 69
        (   L   j   a   v   a   /   l   a   n   g   /   S   t   r   i

                      e
000050 6e 67 3b 29 56|01|00 16|28 5b 4c 6a 61 76 61 2f
        n   g   ;   )   V 001  \0 026   (   [   L   j   a   v   a   /
```

A konstansterület további elemei egészen az utolsó, 31. elemig Utf8 sztringek. (Ezek közül néhány aláírássztringet tartalmaz, mint a 0x0c. is.)

Így a konstansterület valamennyi elemét elolvastuk, a következő eredményt kapva:

- 0x01: sztring, szövege a 0x13. bejegyzésben,
- 0x02: class, neve "HelloVilag",
- 0x03: class, neve "java/lang/PrintStream",
- 0x04: class, neve "java/lang/Object",
- 0x05: class, neve "java/lang/System",
- 0x06: metódus, a tartalmazó osztály neve "java/lang/Object", a metódus neve "<init>", aláírása "()V",
- 0x07: mező a "java/lang/System" osztályból, neve "out", típusa: "Ljava/io/PrintStream",

- 0x08: metódus a "java/lang/PrintStream" osztályból, neve "println", aláírása "(Ljava/lang/String;)V", vagyis ez a println(String s) metódus,
- 0x09: valaminek a neve: "<init>", aláírása "()V", vagyis ez egy paraméter nélküli konstruktor,
- 0x0a: valaminek a neve: "out", típusa: "Ljava/io/PrintStream;"
- 0x0b: valaminek a neve: "println", aláírása "(Ljava/lang/String;)V", vagyis ez a valami egy String paramétert váró, void visszatérési értékű, println nevű metódus lesz,
- 0x0c: Utf8 szöveg: "()V"
- 0x0d: "(Ljava/lang/String;)V"
- 0x0e: "([Ljava/lang/String;)V"
- 0x0f: "<init>"
- 0x10: "Code"
- 0x11: "ConstantValue"
- 0x12: "Exceptions"
- 0x13: "Hello vilag!"
- 0x14: "HelloVilag"
- 0x15: "HelloVilag.java"
- 0x16: "LineNumberTable"
- 0x17: "Ljava/io/PrintStream;"
- 0x18: "LocalVariables"
- 0x19: "SourceFile"
- 0x1a: "java/io/PrintStream"
- 0x1b: "java/lang/Object"
- 0x1c: "java/lang/System"
- 0x1d: "main"
- 0x1e: "out"
- 0x1f: "println"

Ezzel elolvastuk a konstansterületet, most nézzük tovább a ClassFile struktúra szerint a fájl bájtjait:

```
                access this super interf field methods
000160 6e 74 6c 6e|00 21|00 02|00 04|00 00|00 00|00 02|
        n   t   l   n  \0   !  \0 002  \0 004  \0  \0  \0  \0  \0 002
```

access flags

A konstansterület után a .class fájl következő eleme: u2 access_flags

Ez jelen esetben 0x0021 értékű, aminek jelentése, PUBLIC és SYNCHRONIZED, vagyis mindenki számára látható és szinkronizált elérést biztosít. Az egyes bitek beállításával a következő módosítókat adhatjuk meg:

Módosító neve	Értéke	Mező	Metódus	Attribútum
ACC_PUBLIC	0x0001	X	X	X
ACC_PRIVATE	0x0002	X	X	X
ACC_PROTECTED	0x0004	X	X	X
ACC_STATIC	0x0008	X	X	X
ACC_FINAL	0x0010	X	X	X
ACC_SYNCRONIZED	0x0020		X	
ACC_VOLATILE	0x0040	X		
ACC_TRANSIENT	0x0080	X		
ACC_NATIVE	0x0100		X	
ACC_INTERFACE	0x0200			X
ACC_ABSTRACT	0x0400		X	X
ACC_STRICT	0x0800		X	

this, super

A következő két-két bájt megadja a konstansterületen az aktuális és a szülő osztály indexét, esetünkben ezek a "HelloVilag" és "java/lang/Object" osztályok.

interfaces, fields

Két-két bájtban találjuk az osztályban található interfészek és mezők számát, a mi esetünkben mindkettő 0.

```
                  access this super interf field methods
000160 6e 74 6c 6e|00 21|00 02|00 04|00 00|00 00|00 02|
        n   t   l   n  \0   !  \0 002  \0 004  \0  \0  \0  \0  \0 002

       access name  sign attrib name    length   stack
000170 00 09|00 1d|00 0e|00 01|00 10|00 00 00 25|00 02|
        \0  \t  \0 035  \0 016  \0 001  \0 020  \0  \0  \0   %  \0 002
```

methods

A következő 0x0002 jelzi, hogy két metódus van ebben az osztályban:

```
method_info {
      u2 access_flags;
      u2 name_index;
      u2 signature_index;
      u2 attributes_count;
      attribute_info attributes[attributes_count];
      }
```

Az első metódus STATIC és PUBLIC attribútumokkal rendelkezik, neve "main", aláírása "([Ljava/lang/String;)V", vagyis a következőképpen nézhetett ki a forrásprogramban:

```
public static void main(String[] args)
```

A metódushoz tartozik még egy attribútum is (a következő két bájtban 0x0001 az attribútumok száma). A JVM specifikációja jelenleg csak kétféle metódus-attribútumot ismer: "Code" és "Exceptions". Az attribútum első két bájtja egy konstansterületbeli index, az index által mutatott helyen találjuk az attribútum nevét. Jelen esetben ez "Code".

```
Code_attribute {
     u2 attribute_name_index;
     u4 attribute_length;
     u2 max_stack;
     u2 max_locals;
     u4 code_length;
     u1 code[code_length];
     u2 exception_table_length;
     { u2 start_pc;
       u2 end_pc;
       u2 handler_pc;
       u2 catch_type;
     } exception_table[exception_table_length];
     u2 attributes_count;
     attribute_info attributes[attribute_count];
}
```

A constant_pool[attribute_name_index] tehát "Code", a következő 4 bájt az attribútum teljes hossza, a hossz utáni bájttól kezdődően. Példánkban ez 0x25, vagyis 37 további bájt.

```
       access name  sign attrib name    length   stack
000170 00 09|00 1d|00 0e|00 01|00 10|00 00 00 25|00 02|
        \0  \t  \0 035  \0 016  \0 001  \0 020  \0  \0  \0   %  \0 002

       locals codelength|getstat07 ldc1 invvirt ret|exc. table length
000180 00 01|00 00 00 09|b2 00 07 12 01 b6 00 08 b1|00
        \0 001  \0  \0  \0  \t 262  \0  \a 022 001 266  \0  \b 261  \0

          attrib name   length
000190 00|00 01|00 16|00 00 00 0a|00 02 00 00 00 03 00
        \0  \0 001  \0 026  \0  \0  \0  \n  \0 002  \0  \0  \0 003  \0
```

Ezután 2-2 bájtban az operandusverem és a lokális változók területének maximális mérete következik, ez 2, illetve 1 bejegyzés a példánkban. Ezután 4 bájt a tényleges gépi kód hossza, majd következik maga a kód,[3] esetünkben 9 bájt. Ennek a 9 bájtnak a jelentését a következő alfejezetben tárgyaljuk.

A kód után látható, hogy ebben a metódusban nem használunk kivételeket, valamint hogy a kódnak egy attribútuma van. (Nem összekeverendők az osztály, a metódus és a kód attribútumai.) Jelenleg a kódhoz kétféle attribútum adható meg: "LineNumberTable" és "LocalVariables", mindkettő nyomkövetési információkat tartalmaz. Példaprogramunk a "LineNumberTable" attribútumot tartalmazza a main metódusnál, ez elfoglalja a metódus hátralévő bájtjait.

Példánk még az "<init>" nevű metódust tartalmazza, ennek elemzését az Olvasóra bízzuk.

SourceFile

Példaprogramunk listájának utolsó két sorában találjuk a .class fájl struktúra utolsó elemét, az osztályhoz kapcsolódó opcionális attribútumokat. Osztály szintű attribútum jelenleg csak egy lehetséges, ezt találjuk itt is. A "SourceFile" attribútum megadja annak a fájlnak a nevét, amelyből ezt a .class fájlt előállították. Felépítése hasonló az eddig

[3] Kódon a program logikáját (utasításainak hatását) leíró részt értjük.

megismert attribútumokéhoz: 2 bájton szerepel az attribútum nevének konstansterületbeli indexe, utána 4 bájton következik az attribútum további hossza, ez a "SourceFile" attribútum esetén kötelezően 2, végül két bájtban a forrásfájl nevének konstansterületbeli indexe zárja példafájlunkat. Innen megállapíthatjuk, hogy a forrásprogram neve "HelloVilag.java" volt.

Az attribútumokra általános szabály, hogy ha az adott JVM nem tudja értelmezni az attribútumot, akkor azt figyelmen kívül kell hagyni. Ez mindig megtehető, mivel minden attribútum elején a neve utáni 4 bájton a további hossza van megadva:

```
GenericAttribute_info {
      u2 attribute_name;
      u4 attribute_length;
      u1 info[attribute_length];
      }
```

A példaként használt Java program forráskódja a következő:

```
public class HelloVilag {
 public static void main(String[] args) {
    System.out.println("Hello vilag!");
 }
}
```

Az Olvasónak ajánljuk, hogy gyakorlásképpen más lefordított programokat is próbáljon visszafejteni – segítségül felhasználva az od vagy más hasonló programot.

Q.4. Utasítások

A JVM gépi kód utasításainak majdnem mindegyike az operandusvermet használja a műveletek elvégzéséhez. Az utasítások az operandusaikat a verem tetejéről olvassák ki, elvégzik a műveletet és az eredményt a vermen helyezik el. A felhasznált operandusokat általában le is veszik a veremről. Ha szükségünk van további utasításokhoz is a vermen lévő adatokra, akkor azokat explicit külön utasításokkal meg kell őriznünk.

A veremhasználat egyben azt is jelenti, hogy a JVM lényegében nem használ regisztereket, mint ahogyan az sok processzornál szokás, mivel az egyes konkrét processzorokon nagyon eltérő lehet a regiszterek száma és szervezése.

Minden utasítás használja és módosítja viszont a PC (utasításszámláló) regisztert. Ez a következő végrehajtandó utasítás címét tartalmazza. Az utasítások automatikusan megnövelik a PC regiszter értékét, hogy a következő végrehajtandó utasításra mutasson. Természetesen a vezérlésátadó utasítások, mint például a goto, feladatuknak megfelelően módosítják PC értékét.

Amint azt a fejezet elején a JVM általános ismertetésekor említettük, a JVM megköveteli, hogy a veremben és a lokális változók területén lévő adatokra vonatkozó hivatkozásnál a típusuknak megfelelő utasításokat használjuk. A legtöbb esetben az utasítás neve utal arra, hogy milyen típusú adatokra alkalmazható az adott utasítás. Így például az iadd, ladd, fadd, dadd utasítások int, long, float, illetve double típusú számokat adnak össze, és ilyen típusú eredményt adnak. Az "a" betűvel kezdődő nevű utasítások gyakran objektumokra mutató referenciákkal (tömbökkel) dolgoznak.

Az utasítások kódja mindig egyetlen bájt, ez után következhet az utasítástól függően néhány további bájt a kódban, amely meghatározza, hogy milyen értékekkel kell az utasítást végrehajtani. Például a bipush utasítás a kódban utána következő egyetlen bájtot int típusúvá alakítja és elhelyezi a verem tetején.

A következőkben feladatuk szerint csoportosítva bemutatjuk röviden az egyes utasításokat, majd egy példával bemutatjuk ezek használatát.

Konstansok elhelyezése a veremben

- bipush, sipush: az utánuk következő 1, illetve 2 bájtot egész (int) típusúvá alakítva elhelyezik a veremben.
- ldc, ldc_w, ldc2_w: az utasítás után következő egy vagy két bájtot a konstansterület indexeként értelmezik, és az általa mutatott értéket elhelyezik a veremben.
- aconst_null, iconst_m1, ioconst_n, lconst_n, fconst_n, dconst_n: megfelelő típusú konstanst helyeznek el a veremben. Az iconst_n és hasonló utasítások valójában több utasítást jelentenek n=0,1,2,3,4,5 értékekre. (n lehetséges értékei típustól függően eltérőek.)

Lokális változók elhelyezése a veremben

- Xload, Xload_n: X=i,l,f,d vagy a, ezek az utasítások az utána következő bájtban megadott sorszámú lokális változót helyezik el a veremben.

Veremtető értékek elhelyezése lokális változókban

- Xstore, Xstore_n: X=i[4],l,f,d,a.: az utasítás után következő bájtban megadott sorszámú lokális változóba másolja a verem tetejének tartalmát.

Lokális változó növelése

- iinc: az utasítás után következően megadott lokális változót megnöveli a kódban utána következő értékkel. (Tehát NEM a vermet használja.)

Index bitszélességének növelése

- wide: load, store vagy inc utasítások előtt prefixként használva a megadott bájttal együtt az őt követő utasításban megadott lokális változó eléréséhez 16 bites címet használ.

Tömbkezelés

- newarray, anewarray, multianewarray, Xaload, Xastore: tömböt hoz létre, vagy egy megadott tömb adott indexű elemét írja vagy olvassa. A tömböt és a kívánt indexet a vermen kell megadni.

Üres utasítás

- nop: nem csinál semmit.

Veremkezelő műveletek

- pop, pop2: leveszi és eldobja a verem tetején lévő egy vagy két szót (a vermen 32 bites szavakat tárolunk).
- dup, dup2, dup_x1, dup2_x1, dup2_x2: megdupláz egy vagy két szót a vermen; az x-es utasítások az eredményt nem a verem tetejére, hanem két vagy három szóval lejjebb helyezik el.
- swap: megcseréli a verem tetején lévő két szót.

[4] int típust több esetben is használunk, például byte, char és short esetén.

Aritmetikai utasítások

- Xadd, Xsub, Xmul, Xdiv, Xrem, Xneg: a megfelelő műveletet végzik el, X= i, l, f vagy d.

Logikai utasítások

- XshY, Xor, Xxor: X=i, l, Y=l, r logikai léptetés iránya.

Konverziós műveletek

- X2Y: X típusról Y típusra alakít. X=i, l, f, d, int Y=i, l, f, d, b, c, s.

Vezérlésátadó és összehasonlító utasítások

- ifX, if_icmpX, if_acmpX, ifnull, ifnonnull: X=eq, lt, le, ne, gt, ge. Összehasonlítás és ugrás, ha igaz az eredmény.
- XcmpY: X=i, l, f, d Y=semmi, l, g. Összehasonlítást végez, majd az eredménytől függően -1, 0 vagy 1-et tesz a veremre.
- goto, goto_w: feltétel nélküli ugrás (w 4 bájtos, a w nélküli 2 bájtos címet használ).
- jsr, jsr_w: szubrutint hív.
- ret: visszatér szubrutinból.

Metódusból visszatérés

- return, Xreturn: X=i, l, f, d, a void, vagy a nevében megadott típusú értékkel tér vissza a hívóhoz.
- breakpoint: megáll, és a nyomkövető (debug) kezelőnek adja a vezérlést.

Táblázatos ugrás

- tableswitch: az utasítás után felsorolt táblázatnak megfelelően ugrik a verem tetején levő indexet felhasználva.
- lookupswitch: az utasítás után felsorolt értékekkel hasonlítja össze a verem tetején lévő kulcsértéket. Az első egyezésnél megadott eltolás szerint vezérlésátadást végez.

Objektum mezők elérése

- putfield, getfield, putstatic, getstatic: lekérdezi, illetve beállítja egy objektum vagy egy osztály mezőjének értékét.

Metódushívás

- invokevirtual: egy objektum egy metódusát hívja meg, futási időben kiválasztva az objektum típusa szerint. Ez a szokásos Java metódushívás.
- invokespecial: metódushívás az objektum fordítási idejű típusa alapján.
- invokestatic: egy osztály valamely static metódusát meghívja.
- invokeinterface: futási időben keresi az objektum által implementált metódusokat, és meghívja a megfelelőt.

Kivételkezelés

- athrow: kivételt vagy hibát vált ki, keresi a megfelelő catch szerkezetet, ahol a kivételt le lehet kezelni.

Objektumkezelő műveletek

- new: új objektumot hoz létre.
- checkcast: ellenőrzi, hogy az objektum a megadott típusú-e. Ha nem, kivételt vált ki.
- instanceof: ellenőrzi, hogy az objektum a megadott típusú-e. Az eredménytől függően 1-et vagy 0-t ír a veremre.

Monitorok

- monitorenter: megkísérel kizárólagos hozzáférést szerezni a megadott objektumhoz egy lock (zár) mechanizmus segítségével. Ha egy másik thread már foglalja az objektum lockját, akkor várakoztatja az aktuális threadet, amíg a lock felszabadul. Ha a lock még nem foglalt, akkor kizárólagosan lefoglalja. Ha az objektum már rendelkezik a lockkal, akkor nincs hatása.
- monitorexit: elengedi az objektumhoz rendelt lockot és lehetővé teszi, hogy az objektumra váró más threadek továbbhaladjanak. Egy thread több lockot is tarthat ugyanazon az objektumon.

Egyéb

- Történelmi okokból a 0xba kódot nem használják.
- impdep1 és impdep2: e két uasítás kódja rendre 0xfe, illetve 0xff; implementációfüggő műveletekhez használatosak.

Q.4.1. Példa

Példaként nézzük meg az előző alfejezetben bemutatott .class fájlból a main() metódus kódját:

- B20008 get_static 0x0008 a konstansterület 8. elemét tegyük a verembe. Ez egy PrintStream típusú "out" nevű mező.
- 1201 ldc1 01 a konstansterület 1. elemét tegyük a verembe: ez egy sztring, értéke "Hello vilag!"
- b60006 invokevirtual meghívjuk a verem tetején lévő objektumra a veremben lejjebb található paraméterekkel a konstansterület 6. bejegyzésében szereplő metódust. Ez a println(String s) metódus, amely ki fogja írni a sztringben lévő üdvözletet.
- b1 return void értékkel visszatérünk a hívóhoz.

Látható tehát, hogy ez a kód valóban a "Hello vilag!" szöveget írja ki.

Q.5. A Java virtuális gép gépi utasításainak kódjai

A táblázatban szereplő több sorba írt utasításneveket egybe kell olvasni. Például a 23H kódú utasítás neve: fload_1

	0	1	2	3	4	5	6	7
0	nop	aconst_null	iconst_m1	iconst_0	iconst_1	iconst_2	iconst_3	iconst_4
1	bipush	sipush	ldc	ldc_w	ldc2_w	iload	lload	fload
2	lload_2	lload_3	fload_0	fload_1	fload_2	fload_3	dload_0	dload_1
3	faload	daload	aaload	baload	caload	saload	istore	lstore
4	lstore_1	lstore_2	lstore_3	fstore_0	fstore_1	fstore_2	fstore_3	dstore_0
5	lastore	fastore	dastore	aastore	bastore	castore	sastore	pop
6	iadd	ladd	fadd	dadd	isub	lsub	fsub	dsub
7	irem	lrem	frem	drem	ineg	lneg	fneg	dneg
8	ior	lor	ixor	lxor	iinc	i2l	i2f	i2d
9	d2f	i2b	i2c	i2s	lcmp	fcmpl	fcmpg	dcmpl
A	if_icmpne	if_icmplt	if_icmpge	if_icmpgt	if_icmple	if_acmpeq	if_acmpne	goto
B	areturn	return	getstatic	putstatic	getfield	putfield	invokevirtual	invokespecial
C	checkcast	instanceof	monitorenter	monitorexit	wide	multianewarray	ifnull	ifnonnull
D								
E								
F								
	0	1	2	3	4	5	6	7

8	9	A	B	C	D	E	F	
i const_ 5	l const_ 0	l const_ 1	f const_ 0	f const_ 1	f const_ 2	d const_ 0	d const_ 1	0
d load	a load	i load_ 0	i load_ 1	i load_ 2	i load_ 3	l load_ 0	l load_ 1	1
d load_ 2	d load_ 3	a load_ 0	a load_ 1	a load_ 2	a load_ 3	ia load	la load	2
f store	d store	a store	i load_ 0	i load_ 1	i load_ 2	i load_ 3	l store_ 0	3
d store_ 1	d store_ 2	d store_ 3	a store_ 0	a store_ 1	a store_ 2	a store_ 3	ia store_	4
pop 2	dup	dup_ x1	dup_ x2	dup2	dup2_ x1	dup2_ x2	swap	5
i mul	l mul	f mul	d mul	i div	l div	f div	d div	6
i sh l	l sh l	i sh r	l sh r	iu sh r	lu sh r	i and	l and	7
l 2 i	l 2 f	l 2 d	f 2 i	f 2 l	f 2 d	d 2 i	d 2 l	8
d cmp g	if eq	if ne	if lt	if ge	if gt	if le	if_ icmp eq	9
jsr	ret	table switch	lookup switch	i return	l return	f return	d return	A
invoke static	invoke inter face		new	new array	a new array	array length	a throw	B
goto_ w	jsr_ w	break point						C
								D
								E
						imp dep 1	imp dep 2	F
8	9	A	B	C	D	E	F	