# Programiranje z javansko zložno kodo

**Andraž Drčar, Tomaž Dobravec**

*Univerza v Ljubljani, Fakulteta za računalništvo in informatiko*
*Tržaška 25, Ljubljana, Slovenija*
*E-pošta: andraz.drcar@gmail.com*

## Java inline bytecode

*Java is one of the top programming languages known for its platform independency which is reached by using platform specific Java Virtual Machines (JVM). Each JVM follows strict rules how class files containing the bytecode are parsed and executed. However, there are no such rules for the compilation part.*

*This article describes the structure of the class file as well as the solution of how to implement a Java compiler that allows usage of inline bytecode. Such a compiler could be useful for educational purposes to present the bytecode mechanics or some specific instructions. Advanced programmers that already understand the background of Java could also use this compiler to optimize the key segments of their code.*

## 1 Uvod

Java je eden izmed bolj razširjenih programskih jezikov, ki je znan predvsem po svoji sistemski neodvisnosti. To pomeni, da prevedene kode po selitvi na nek drug sistem ni potrebno popravljati ali ponovno prevajati. Za to poskrbi Javansko okolje s svojim navideznim strojem (*angl. Java Virtual Machine - JVM*), ki je razvit za vrsto različnih procesorjev in operacijskih sistemov. Vsi ti stroji sledijo natančnim pravilom, kako brati in izvajati prevedeno kodo, ki je zapisana v datotekah tipa `class` kot zaporedje ukazov zložne kode. Med tem, ko je izvajanje natančno predpisano, pa za prevajanje ne obstajajo standardizirana pravila. Ker obstaja veliko število različnih sistemov, obstaja tudi veliko prevajalnikov različnih razvijalcev. Obstaja torej možnost, da se isti program prevede v različno zaporedje ukazov zložne kode. Posledica tega je, da programer nima nikakršnega vpliva na prevedeno datoteko in nima možnosti za optimizacijo.

V tem članku je opisan prevajalnik, ki omogoča uporabo zložne kode pri programiranju v Javi s pomočjo novega ukaza `jasm`. Takšen prevajalnik omogoča programerju optimizacijo prevedene kode, saj so ključni deli lahko napisani direktno z ukazi zložne kode. Poleg optimizacije se lahko prevajalnik uporablja tudi v izobraževalne namene. Predstavimo lahko splošno delovanje Jave, uporabo sklada pri navideznem stroju ter tudi specifične ukaze javanske zložne kode. Preden pa se lotimo predstavitve delovanja prevajalnika, moramo spoznati še nekaj osnov delovanja virtualnega stroja, predvsem pa samo sestavo prevedene datoteke.

## 2 JVM in predstavitev zložne kode

JVM je neke vrste navidezni procesor s svojim naborom ukazov. Sprva je bil namenjen izvajanju prevedene kode programskega jezika Java, kasneje pa so se pojavili tudi prevajalniki, ki tvorijo zložno kodo za JVM tudi iz drugih programskih jezikov. Skupaj z javanskim programskim vmesnikom in naborom standardnih knjižnic JVM tvori javansko izvajalno okolje (angl. Java Runtime Environment - JRE) [1, 2].

JVM je sestavljen iz enote za preverjanje zložne kode, enote za delo in upravljanje s pomnilnikom ter tolmača (*angl. interpreter*). Enota za preverjanje zložne kode najprej preveri, da vsi skoki znotraj programa kažejo na veljavno lokacijo ter da so vse spremenljivke ustrezno inicializirane. Med samim izvajanjem skrbi tudi za dostope do zaščitenih delov kode. Kot pomnilnik JVM uporablja kopico, kamor se shranijo vsi objekti - primerki razredov. Za čiščenje tega dela pomnilnika skrbi čistilec pomnilnika (*angl. garbage collector*). V preostalem delu pomnilnika se nahajajo tabele konstant, kode metod ter po en sklad za vsako izvajajočo se nit. Zadnja komponenta je tolmač, ki ukaze zložne kode prevede in posreduje dejanskemu procesorju sistema. Od leta 2014 se v JVM za doseganje večje hitrosti izvajanja po večini uporablja dinamično prevajanje [3]. To pomeni, da se zložna koda prevaja v ukaze za ustrezno platformo med samim izvajanjem.

Dejstvo, da JVM deluje kot skladovni avtomat, nam pomaga pri razumevanju ukazov zložne kode. Ukazi se v prevedeni datoteki nahajajo v blokih z opisom metod v parametru z imenom `code`. Za razvoj prevajalnika s podporo programiranja v zložni kodi je pomembno, da poznamo sestavo prevedene datoteke. JVM prevedeno datoteko bere in analizira po vrsti, saj le ta nima nobenih naslovov. Enako je potrebno storiti v prevajalniku, s tem da nepotrebno kodo enostavno preskočimo.

### 2.1 Sestava datoteke z zložno kodo

Vsaka `class` datoteka se prične z zaporedjem štirih bajtov in sicer `0xCAFEBABE`. S tem se JVM sporoči, da gre za veljavno `class` datoteko, je pa to izmišljen niz avtorja. Sledijo podatki o verziji in podverziji, ki prav tako zasedejo štiri bajte. Nato imamo podatke o konstantah, vmesnikih, poljih, metodah ter atributih. Povsod je najprej podano število vsebovanih elementov, nato pa opisi samih elementov. Osnovna zgradba je prikazana v tabeli 1.

Tabela 1. Osnovna zgradba datoteke z zložno kodo

| | |
|---|---|
| CAFEBABE | 4B |
| podatki o verziji in podverziji | 4B |
| število constant | 2B |
| deskriptor konstant | |
| zastavice dostopa | 2B |
| razred | 2B |
| nadrazred | 2B |
| število vmesnikov | 2B |
| deskriptor vmesnikov | |
| število polj | 2B |
| deskriptor polj | |
| število metod | 2B |
| deskriptor metod | |
| število atrbutov | 2B |
| deskriptor atributov | |

Podrobneje si bomo ogledali sestavo podatkov, ki so pomembni za delovanje prevajalnika. Deskriptor konstant za vsako konstanto vsebuje podatek o tipu ter vrednost. Vsi tipi in velikosti njihovih vrednosti so zapisani v tabeli 2. Pri prvem tipu, ki predstavlja znakovne konstante, nam prva dva bajta podatkov povesta, kako dolg je niz. Sledijo jima podatki niza.

Tabela 2. Vrste konstant z opisom.

| Tip | Velikost | Opis |
|---|---|---|
| 1 | 2+x B | UTF-8 (Unicode) znakovna konstanta |
| 3 | 4 B | Integer |
| 4 | 4 B | Float |
| 5 | 8 B | Long |
| 6 | 8 B | Double |
| 7 | 2 B | Referenca na razred |
| 8 | 2 B | Reference na znakovno konstanto |
| 9 | 4 B | Referenca na polje |
| 10 | 4 B | Referenca na metodo |
| 11 | 4 B | Referenca na vmesnik |
| 12 | 4 B | Deskriptor z imenom in tipom |

Sestava deskriptorja metod je prikazana v tabeli 3. Za prevajalnik sta najbolj pomembna atribut z vsebino kode in pa atribut s tabelo lokalnih spremenljivk.

Tabela 3. Sestava deskriptorja metod.

| | |
|---|---|
| zastavice dostopa | 2B |
| indeks imena | 2B |
| deskriptor | 2B |
| število atributov | 2B |
| deskriptor atributov | |

V deskriptorju atributov deskriptorja metod se pojavita atribut z opisom kode in atribut z opisom lokalnih spremenljivk, ki sta pomembna za delovanje prevajalnika.

Da lahko prevajalnik preskoči nepomembne dele prevedene datoteke, je potrebno poznati podrobno sestavo vseh deskriptorjev, ki pa si jih bralec lahko ogleda v Oraclovi dokumentaciji [1].

### 2.2 Ukazi zložne kode

Ukazi zložne kode so vsi dolgi en bajt, kar nam omogoča 256 ukazov. Ker JVM uporablja sklad, je veliko ukazov brez dodatnih operandov. K temu pripomore tudi dejstvo, da so nekateri ukazi skrajšani. Na primer, namesto ukaza `iload` z operandom 1, lahko uporabimo skrajšani ukaz `iload_1`. Oba ukaza sta sprejemljiva in oba na sklad naložita vrednost lokalne spremenljivke z indeksom 1. Tovrstni ukazi so bili uporabljeni pri izvedbi prevajalnika.

## 3 Zahteve novega prevajalnika

Kot že rečeno, je bila ideja izdelati prevajalnik, ki bo omogočal uporabo bloka `jasm` z ukazi Javanske zložne kode. Ti ukazi naj bi se ob prevodu skopirali in vstavili na ustrezno mesto v prevedeno datoteko. Način programiranja z uporabo `jasm` bloka si najlažje ogledamo na preprostem primeru.

Koda 1. Primer metode napisane v programskem jeziku Java

```
public static void main(String[] args) {
    int a = 1;
    int b = 3;
    int c = 16;
    int d = a + b + c;
    System.out.println(d);
}
```

Koda 2. Ukazi zložne kode v katere se prevede zgornja koda

```
public static void main(java.lang.String[] args) {
    iconst_1;
    istore_1;           /* a */
    iconst_3;
    istore_2;           /* b */
    bipush 16;
    istore_3;           /* c */
    iload_1;            /* a */
    iload_2;            /* b */
    iadd;
    iload_3;            /* c */
    iadd;
    istore 4;           /* d */
    getstatic 16;       /* java.lang.System.out */
    iload 4;            /* d */
    invokevirtual 22;   /* void println(int b) */
    return;
}
```

Koda 3. Primer uporabe `jasm` blokov z istim rezultatom

```
public static void main(String[] args) {
    int a = 0;
    int b = 0;
    int c = 16;
    jasm {
        iconst_1;
        istore a;
        bipush 3;
        istore b;
    }
    int d = a + b;
    jasm {
        iload d;
        iload c;
        iadd;
        istore d;
    }
    System.out.println(d);
}
```

V kodi 2 je razvidno, kako se javanska koda iz kode 1 prevede v ukaze zložne kode. Koda 3 pa prikazuje uporabo `jasm` bloka, ki ustvari enako zložno kodo kot javanski program v kodi 1. Vsi zgoraj zapisani primeri kot rezultat na zaslon izpišejo številko 20. Kot vidimo so ukazi znotraj `jasm` bloka ločeni s podpičji, argumenti pa so od ukaza in med seboj ločeni s presledki. Verjetno ni odveč omeniti tudi to, da so ukazi nekoliko prirejeni, saj za argumente omogočajo uporabo imen lokalnih spremenljivk. Znotraj programa kot tudi znotraj posamezne metode je mogoče uporabiti več `jasm` blokov. Prikazano je tudi, da programerju ni potrebno poznati vseh skrajšanih ukazov, saj je delovanje ukaza `bipush 3`, ki na sklad potisne vrednost 3 ekvivalentno skrajšanemu ukazu `iconst_3`.

# 4 Opis rešitve

Ko se dodobra spoznamo s sestavo prevedenih datotek, je izvedba prevajalnika precej preprosta. Večina dela se skriva v tem, da analiziramo prevedene datoteke, ki so bile tako ali drugače spremenjene. A kljub vsemu je potrebno biti pozoren na določene stvari, ki so v nadaljevanju opisane pri posameznih delih prevajalnika. Prevajalnik je v grobem razdeljen na naslednjih pet delov:

- Priprava datotek
- Analiza prevedene datoteke
- Določitev mesta in analiza kode
- Rezervacija prostora
- Vstavljanje kode in optimizacija

## 4.1 Priprava datotek

Ker bo potrebno izvorno kodo za delovanje prevajalnika spreminjati, je najprej potrebno ustvariti varnostno kopijo izvorne datoteke, čemur sledi čiščenje izvorne kode. Odstranimo vse komentarje in si označimo mesta `jasm` blokov. To storimo tako, da jih zamenjamo z nizom znakov, ki se ne pojavi nikjer drugje v programu. Pred tem si vsebino `jasm` blokov shranimo. Glede na to, da lahko datoteka še vedno vsebuje znakovne spremenljivke, je potrebno izbrati takšno zaporedje znakov, ki tvori neveljavno spremenljivko tipa `String`. Takšen primer je zaporedje `j\d`, saj `\d` predstavlja neveljavno ubežno sekvenco, črka pred njo pa poskrbi, da bi to bila edina ubežna sekvenca v nizu znakov. To zaporedje znakov bomo v nadaljevanju v različnih datotekah zamenjali z različno kodo, kar je potrebno za delovanje posameznih delov prevajalnika. Pripravimo si pet kopij tako spremenjene datoteke.

## 4.2 Analiza prevedene datoteke

V prvi datoteki pobrišemo izbrano znakovno zaporedje in dobljeno kodo prevedemo z uporabo Javanskega prevajalnika. Ta koda nam služi kot iztočnica, saj nima nobene dodatne vsebine, ki je potrebna za delovanje prevajalnika, kot bomo videli kasneje. Zato si pri analizi prevedene kode shranimo vse podatke pred in po deskriptorju metod. Prav tako si posebej shranimo tabelo konstant, tako da je možno z indeksom dobiti ime konstante kot tudi obratno. Za čisto osnovno delovanje prevajalnika ta korak niti ni potreben, vendar nam omogoča nek del optimizacije, ki jo bomo izvedli v zadnjem delu prevajanja.

Velja še omeniti, da so vsa vmesna prevajanja pognana z dodatnim parametrom `-g`, s katerim prevajalnik ustvari tudi podatke, ki so sicer potrebni zgolj pri razhroščevanju. Sem sodijo lokalne spremenljivke v tabeli konstant ter tabele lokalnih spremenljivk v atributih deskriptorja metod, ki za izvajanje programa po prevodu dejansko niso potrebne. Te podatke potrebujemo pri povezovanju lokalnih spremenljivk z ustreznimi indeksi in ugotavljanju vidnost lokalnih spremenljivk.

## 4.3 Določitev mesta in analiza kode

Sledi tretji korak prevajanja, ki je od vseh najbolj zahteven, saj v njem naredimo tri stvari. Prva stvar je določanje mesta, kjer se `jasm` blok nahaja. To je najlažje storiti z uporabo dveh pripravljenih datotek, ki jima na mesta `jasm` blokov vpišemo različna ukaza. V prvi datoteki vsa izbrana zaporedja znakov, zamenjamo z ukazom `System.exit(0)`, v drugi pa z `System.exit(1)`. Dobra lastnost izbranega ukaza je v tem, da se prevoda razlikujeta že v prvem bajtu, saj se argument ukaza najprej zapiše na sklad. Pri tem so nam v pomoč skrajšani ukazi, ki zasedejo le en bajt. Pomembno je, da kot parametra uporabimo konstanti, ki imata ustrezni skrajšani ukaz. Ustrezno spremenjeni datoteki prevedemo in ponovno analiziramo. Iz prve datoteke si zopet shranimo tabelo konstant. To je pomembno zato, ker nam dodani ukaz lahko v tabelo konstant doda zapis za klic funkcije `System.exit()`, s čimer nam spremeni številke indeksov konstant glede na prevedeno kodo brez dodanega ukaza. Med analizo obe kodi primerjamo in kjer se kodi razlikujeta, je mesto `jasm` bloka.

Analiza prevedenih datotek je precej časovno potraten postopek, zato se želimo izogniti večkratni analizi. Ker se po analizi kode nahajamo v atributih metode, kjer je tudi tabela lokalnih spremenljivk, je najbolj primerno, da v tem koraku preverimo še vidljivost lokalnih spremenljivk. To nam omogoča da prevajalnik programerja opozori na napako, če je v `jasm` bloku uporabil lokalno spremenljivko, ki ni vidna na mestu, kjer se `jasm` blok nahaja.

Tretja stvar, ki jo naredimo v tem koraku, je izračun velikosti prostora, ki ga potrebujejo ukazi `jasm` bloka. To bi sicer lahko storili že prej, a je izračun na tem mestu bolj optimalen. V tem koraku lahko namreč zamenjamo vse lokalne spremenljivke z indeksi iz tabele lokalnih spremenljivk. Nekatere ukaze z znanimi operandi pa lahko pretvorimo v skrajšane. Tako lahko namesto ukaza `iload 1`, ki zasede dva bajta, uporabimo ukaz `iload_1` velikosti en bajt.

## 4.4 Rezervacija prostora

Najprej bi pomislili, da za vstavljanje novih ukazov ni potrebno drugega, kot vriniti ukaze na pravo mesto in ustrezno povečati bajte s podatkom o dolžini kode. A po

preučitvi načina delovanja ukazov zložne kode ugotovimo, da se za skoke uporabljajo odmiki in ne naslovi. Tako nam argument pri pogojnem ali brezpogojnem skoku pove, koliko ukazov naprej ali nazaj naj se premaknemo med izvajanjem. Ugotavljanje ali se `jasm` blok nahaja v zanki ali zunaj nje, oziroma ali je njegova lokacija nekje med ukazi, ki jih je potrebno preskočiti bi bilo zelo mučno opravilo. Poleg tega bi bilo potrebno poiskati vse možne operacije skokov in ustrezno popraviti njihov odmik.

Izkaže se, da je veliko bolj preprosto rezerevirati prostor, ki ga koda bloka potrebuje. Velikost bloka smo že izračunali v prejšnjem koraku. Vzamemo še zadnji dve datoteki, kjer izbrano zaporedje znakov, ki označuje lokacijo `jasm` bloka, zamenjamo z ukazi, katerih prevedena velikost bo ustrezala prostoru, ki ga potrebujemo za vrinjeno kodo. Da nam ni potrebno šteti ukazov med analizo, kljub temu, da imamo lokacijo že znano, zopet uporabimo trik iz točke 4.3. in uporabimo ukaza `System.exit(0)` ter `System.exit(1)`. Ta ukaz v prevodu zasede štiri bajte. Dokler je velikost bloka večja od rezerviranega prostora, dodajamo ukaze `System.gc()`. Ta ukaz se prevede v kodo dolžine tri bajte in poleg imena metode ne dodaja novih spremenljivk v tabelo konstant. Prav tako nikakor ne vpliva na vrednosti lokalnih spremenljivk in ne spreminja poteka programa.

S to metodo nimamo nobenih problemov z odmiki pri skokih, saj vrinjeni ukazi dejansko zamenjajo ukaze v rezerviranem prostoru. Prav tako so dolžine kode metod že pravilno nastavljene. Ustrezno popravljeni datoteki zopet prevedemo in sledi še zadnji korak prevajalnika.

### 4.5 Vstavljanje kode in optimizacija

Še zadnjič opravimo analizo dveh datotek in primerjamo njuno vsebino. Kjer pride do spremembe zaradi uporabe ukazov `System.exit()` z različnima parametroma, enostavno pričnemo s prepisovanjem obstoječih ukazov z ukazi ustreznega `jasm` bloka. Bloki si sledijo zaporedno, zato na prvo mesto razlike zapišemo ukaze prvega bloka. Pred tem je potrebno ukaze, ki za argumente ne sprejemajo lokalnih spremenljivk, še ustrezno popraviti. Potrebno je zamenjati konstante iz tabele konstant z ustreznimi indeksi in spremeniti številske konstante v pravilen format glede na uporabljen tip spremenljivke in ukaz. V kolikor je bilo rezerviranega več prostora, kot ga je pripadajoči blok potreboval (če dolžina bloka ne ustreza formuli 3n + 4), enostavno dodamo ustrezno število ukazov, ki ne naredijo ničesar - `nop` (angl. no operation). Če kot rezultat enostavno zapišemo dobljeno kodo z vstavljenimi ukazi, že dobimo delujočo `class` datoteko. V tem primeru moramo biti pozorni, da pri popravljanju ukazov uporabimo tabelo konstant iz zadnje analize.

Toda pri sestavljanju končne datoteke lahko tu opravimo nek del optimizacije. To dosežemo tako, da za kodo pred in za blokom z opisi metod uporabimo podatke iz prve datoteke, ki smo jih shranili v prvem koraku. Prav tako je potrebno pri popravljanju ukazov pred vstavljanjem v tem primeru uporabiti ustrezno tabelo lokalnih spremenljivk, saj je le-ta del kode pred opisom metod. V tem primeru je dela nekoliko več, saj moramo opis metod izluščiti iz zadnje datoteke in ga vstaviti med shranjena dela kode. A kot smo že omenili, je rezultat optimalnejša koda, poleg tega pa ta metoda omogoča še nekatere dodatne optimizacije opisane v nadaljevanju.

## 5 Zaključek

V tem delu smo predstavili prevajalnik javanske kode, ki poleg običajnih ukazov omogoča uporabo `jasm` blokov, ki vsebujejo čisto JVM zložno kodo. Osnovno funkcionalnost prevajalnika smo podprli, obstaja pa še veliko možnosti za izboljšavo. Nekatere so enostavnejše, spet druge pa zahtevajo velik poseg v opisani prevajalnik. Prva izboljšava je nadaljnja optimizacija, predvsem v smislu uporabe tabele konstant brez lokalnih spremenljivk. To bi lahko dosegli tako, da bi kot izhodišče vzeli kodo datoteke, prevedene brez parametra `-g`. V tem primeru bi lahko tudi izpustili atribute deskriptorja metod, kjer so shranjeni podatki o lokalnih spremenljivkah. Druga možna izboljšava je uporaba parametrov, ki so dovoljeni pri standardnem prevajanju. Osnova takega prevajanja bi bila datoteko, ki bi jo dobili tako, da bi parametre posredovali prevajalniku. Tu bi bilo potrebno še dopolniti prepisovanje bloka z opisom metod glede na poslane parametre. Naslednja izboljšava bi omogočala dodajanje novih spremenljivk. Prevajalnik bi lahko v primeru uporabe ukaza `istore x` in ob predpostavki, da spremenljivka z imenom `x` še ni bila definirana, ustrezno dopolnil tabelo konstant in zapis dodal še v tabelo lokalnih spremenljivk. Toda tu se pojavi vprašanje, ali naj bo takšna spremenljivka le začasna in se lahko uporablja zgolj znotraj trenutnega `jasm` bloka, ali naj bo vidna tako, kot da je bila definirana na mestu `jasm` bloka. Zadnja in morda najbolj zanimiva nadgradnja pa bi bila izdelava vtičnikov za obstoječa razvojna okolja (Eclipse, Netbeans, ...). V tem primeru bi bilo potrebno zamenjati uporabo standardnega prevajalnika z opisanim prevajalnikom s podporo `jasm` blokov, kot tudi dograditev prepoznavanja razširjene sintakse, s čimer bi omogočili uporabo `jasm` blokov.

## Literatura


[1] Tim Lindholm, Frank Yellin. Java Virtual Machine Specification. Addison-Wesley. Boston, MA. 1999

[2] Joshua Engel. Programming for the Java Virtual Machine. Reading (Massachusetts). Addison-Wesley, 1999.

[3] en.wikipedia.org/wiki/Java_virtual_machine

[4] en.wikipedia.org/wiki/Java_bytecode_instruction_listings