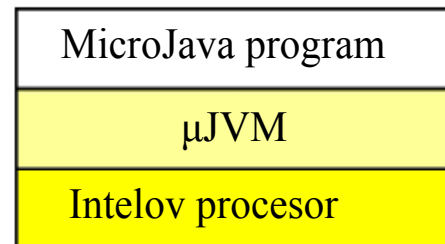


Arhitektura MikroJava virtuelne mašine (μ JVM)

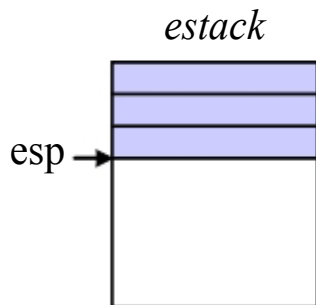
Šta je virtuelna mašina (VM)?

- CPU implementiran u softveru
- instrukcije se interpretiraju (ili "jit-uju")
- primeri: Java VM, Microsoft CLR, Paskalski P-kod



μ JVM je stek mašina

- nema registara
- umesto njih ima *stek izraza* (koji čuva vrednosti koje se računaju)



niz mem. reči (1 reč = 4 bajta)
ne mora biti veliki (npr. 32 words $\approx$ 32 registra)

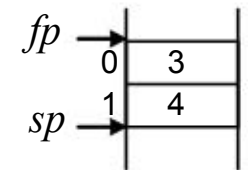
esp ... expression stack pointer

Kako radi Stek Mašina

Primer

iskaz $i = i + j * 5;$

i i j su npr. lokalne promenljive



Izvršavanje mjym programa

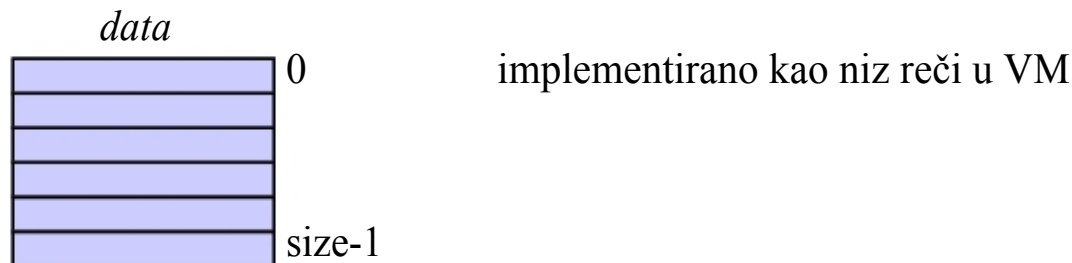
maš.instrukc *e-stek*

load0	3	load variable from address 0 (i.e. i)
load1	34	load variable from address 1 (i.e. j)
const5	345	load constant 5
mul	320	multiply the two topmost stack elements
add	23	add the two topmost stack elements
store0		store the topmost stack element to address 0

Na kraju svakog iskaza e-stek treba da je prazan!

Oblasti podataka μJVM

Globalni podaci

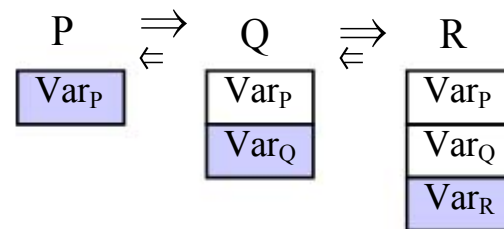
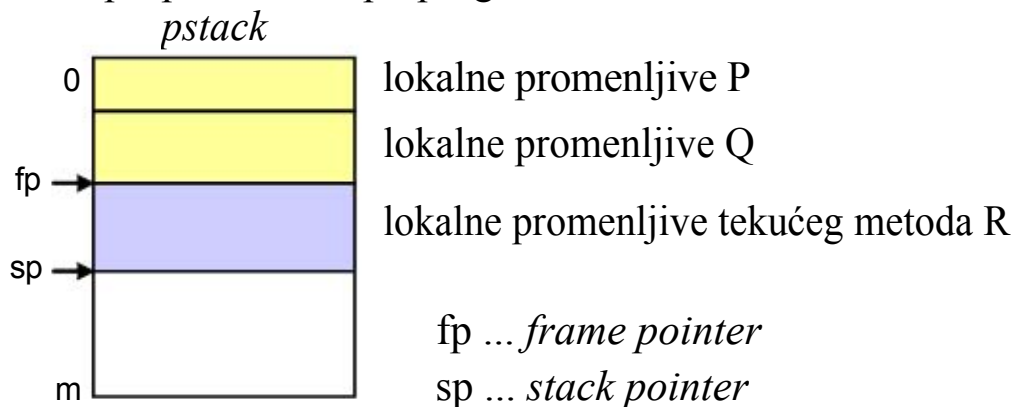


- oblast fiksne veličine
- globalne promenljive postoje tokom čitavog programa
- svaka promenljiva zauzima 1 memorijsku reč (4 bajta)
- adresiraju se uz navođenje indeksa u nizu *data*
npr. *getstatic 2* učitava vrednost sa adrese 2 niza *data* na *e-stek*

Oblasti podataka μJVM

Lokalne promenljive

- alociraju se unutar okvira **programskog** steka
- svaki poziv metoda dobija sopstveni okvir na steku
- okvirima se stavljaju na stek pri pozivu, a skidaju sa steka pri povratku iz potprograma

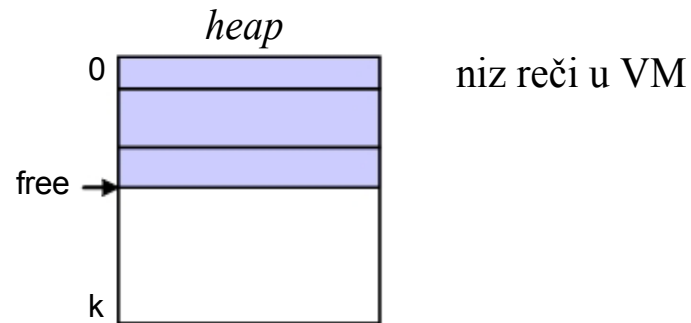


- lokalne promenljive se adresiraju relativno u odnosu na *fp*
- svaka promenljiva zauzima po 1 reč (4 bajta)
- npr. *load0* učitava vrednost promenljive sa adrese *fp+0* na *e-stek*

Oblasti podataka μJVM

Dinamički podaci - Heap

- za čuvanje instanci klasa i nizova

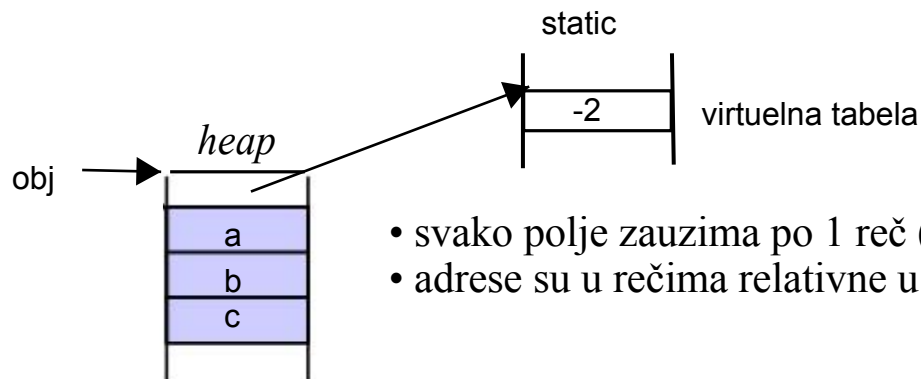


- Novi objekti se alociraju na poziciji *free* (pa se *free* uvećava); ovo se radi VM instrukcijama *new* i *newarray*
- Objekti se nikad ne dealociraju (nema “sakupljanja smeća”- garbage collection)
- Pokazivači su adrese na nivou reči

Oblasti podataka μ JVM

Objekat klase

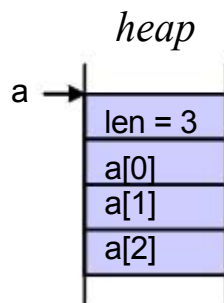
```
class X {  
    int a, b;  
    char c;  
}  
X obj = new X;
```



- svako polje zauzima po 1 reč (4 bajta)
- adrese su u rečima relativne u odnosu na *obj*

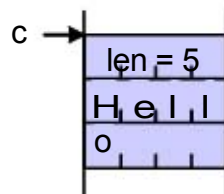
Niz

```
int[] a;  
a = new int[3];
```



- dužina niza se pamti na početku
- svaki element zauzima po 1 reč

```
char[] c = new char[5];
```

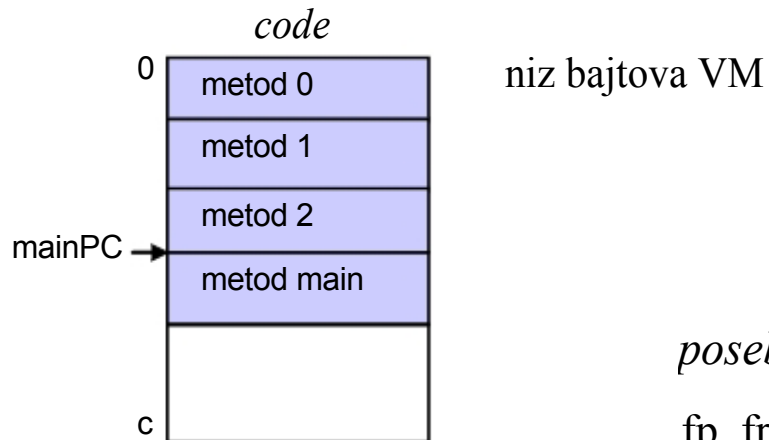


- *char* nizovi su nizovi bajtova
- alociranje prostora je na nivou reči

Oblast koda μJVM

Programski kod

- niz bajtova fiksne veličine
- metodi se alociraju jedan za drugim
- *mainPC* ukazuje na *main()* metod



posebni registeri VM

fp frame pointer
sp stack pointer (pstack)
esp stack pointer (estack)
pc program counter

Skup instrukcija μJVM

Bajtkod (slično bajtkodu Java virtuelne mašine)

- Kompaktan: većina istrukcija od po 1 bajt
- ne pamte se tipovi (kod Java VM tipovi operanada su kodirani u instrukcijama)

MicroJava

load0
load1
add

Java

iload0
iload1
iadd

fload0
fload1
fadd

razlog: Java verifikator bajtkoda
može na osnovu tipova operanada proveriti
integritet programa

Format μJVM instrukcija

Code = {Instruction}.
Instruction = opcode {operand}.

opcode ... 1 bajt
operand ... 1, 2 ili 4 bajta

Primeri

0 operanada	add	ima 2 implicitna operanda na steku
1 operand	load 7	
2 operanda	enter 0, 2	početak metoda

Skup instrukcija μJVM

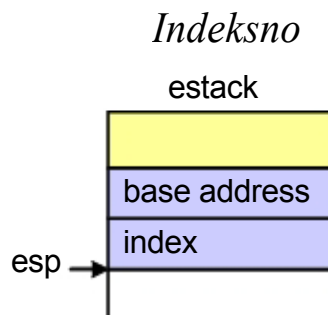
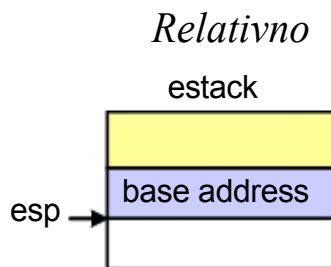
Načini Adresiranja

Kako se može pristupiti operandima?

način adresiranja

primer instrukcije

- | | | |
|---------------------|-------------|--|
| • Neposredno | const 7 | za konstante |
| • Lokalno | load 3 | za lokalne promenljive na <i>psteku</i> |
| • Statičko | getstatic 3 | za globalne promenljive u <i>data</i> oblasti |
| • Stek | add | za vrednosti na <i>esteku</i> |
| • Relativno | getfield 3 | za polja objekata (load $heap[pop() + 3]$) |
| • Indeksno | aload | za elemente nizova (load $heap[pop() + 1 + pop()]$) |



Skup instrukcija μ JVM

Load/store lokalnih promenljivih

load b	..., val	<u>Load</u> push(local[b]);
load <n>	..., val	<u>Load</u> (n = 0..3) push(local[n]);
store b	..., val	<u>Store</u> local[b] = pop();
store <n>	..., val	<u>Store</u> (n = 0..3) local[n] = pop();

dužine operanada

b ... bajt

s ... short (2 bajta)

w ... word (4 bajta)

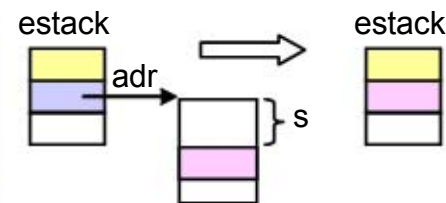
Load/store globalnih promenljivih

getstatic s	..., val	<u>Load static variable</u> push(data[s]);
putstatic s	..., val	<u>Store static variable</u> data[s] = pop();

Skup instrukcija μJVM

Load/store polja objekata

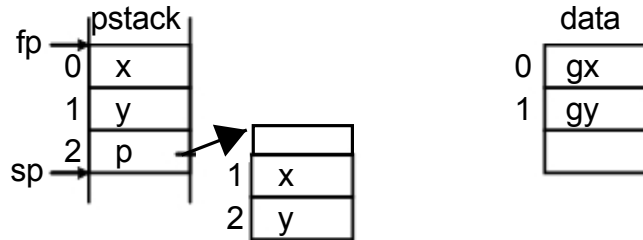
getfield s	..., adr ..., val	<u>Load object field</u> adr = pop(); push(heap[adr+s]);
putfield s	..., adr, val	<u>Store object field</u> val = pop(); adr = pop(); heap[adr+s] = val;



Load konstante

const w	..., val	<u>Load constant</u> push(w);
const<n>	..., val	<u>Load constant</u> (n = 0..5) push(n);
const_m1	..., val	<u>Load minus one</u> push(-1);

Primeri: Load i Store



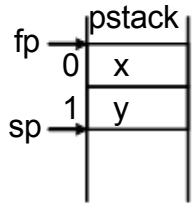
	<i>code</i>	<i>bytes</i>	<i>stack</i>
x = y;	load1 1 store0 1		y -
gx = gy;	getstatic 1 putstatic 0	3 3	gy -
p.x = p.y;	load2 1 load2 1 getfield 2 putfield 1	 3 3	p p p p p.y -

Skup instrukcija μJVM

Aritmetičke instrukcije

add	..., val1, val2 ..., val1+val2	<u>Add</u> push(pop() + pop());
sub	..., val1, val2 ..., val1-val2	<u>Subtract</u> push(-pop() + pop());
mul	..., val1, val2 ..., val1*val2	<u>Multiply</u> push(pop() * pop());
div	..., val1, val2 ..., val1/val2	<u>Divide</u> x = pop(); push(pop() / x);
rem	..., val1, val2 ..., val1%val2	<u>Remainder</u> x = pop(); push(pop() % x);
neg	..., val ..., -val	<u>Negate</u> push(-pop());
shl	..., val, x ..., val1	<u>Shift left</u> x = pop(); push(pop() << x);
shr	..., val, x ..., val1	<u>Shift right</u> x = pop(); push(pop() >> x);

Primeri: Aritmetičke operacije



$x + y * 3$

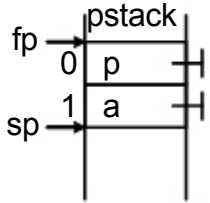
<i>code</i>	<i>bytes</i>	<i>stack</i>
load0 1		x
load1	1	x y
const3	1	x y 3
mul	1	x y*3
add 1		x+y*3

Skup instrukcija μJVM

Kreiranje objekata

new s	..., adr	<u>New object</u> allocate area of s words; initialize area to all 0; push(adr(area));
newarray b	..., n ..., adr	<u>New array</u> n = pop(); if (b == 0) allocate array with n elements of byte size; else if (b == 1) allocate array with n elements of word size; initialize array to all 0; store n as the first word of the array; push(adr(array));

Primeri: Kreiranja objekata

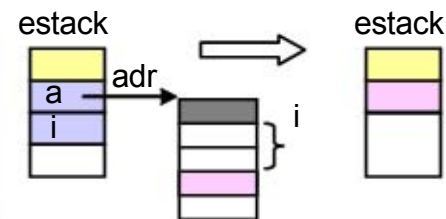


	<i>code</i>	<i>bytes</i>	<i>stack</i>	
Person p = new Person;	new 4 store0 1	3	p -	// assume: size(Person) = 4 words
int[] a = new int[5];	const5 1 newarray 1 store1 1	2	5 a -	

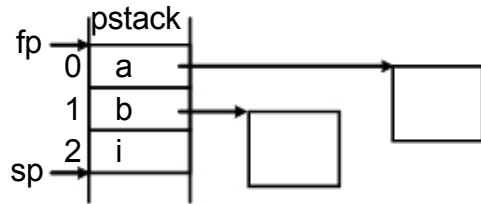
Skup instrukcija μJVM

Pristup nizovima

aload	..., adr, i ..., val	<u>Load array element</u> i = pop(); adr = pop(); push(heap[adr+1+i]);
astore	...,adr, i, val	<u>Store array element</u> val = pop(); i = pop(); adr = pop(); heap[adr+1+i] = val;
baload	..., adr, i ..., val	<u>Load byte array element</u> i = pop(); adr = pop(); x = heap[adr+1+i/4]; push(byte i%4 of x);
bastore	...,adr, i, val	<u>Store byte array element</u> val = pop(); i = pop(); adr = pop(); x = heap[adr+1+i/4]; set byte i%4 in x; heap[adr+1+i/4] = x;
arraylength	..., adr ..., len	<u>Get array length</u> adr = pop(); push(heap[adr]);



Primer: Pristup členu niza



	<i>code</i>	<i>bytes</i>	<i>stack</i>
a[i] = b[i+1];	load0 1		a
	load2	1	a i
	load1	1	a i b
	load2	1	a i b i
	const1 1		a i b i 1
	add	1	a i b i+1
	aload	1	a i b[i+1]
	astore 1		-

Skup instrukcija μJVM

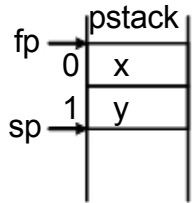
Manipulacija stekom

pop	..., val	<u>Remove topmost stack element</u> dummy = pop();
------------	----------	---

Skokovi

jmp s		<u>Jump unconditionally</u> pc = s;
j<cond> s	..., x, y	<u>Jump conditionally</u> (eq,ne,lt,le,gt,ge) y = pop(); x = pop(); if (x cond y) pc = s;

Primer: Skok



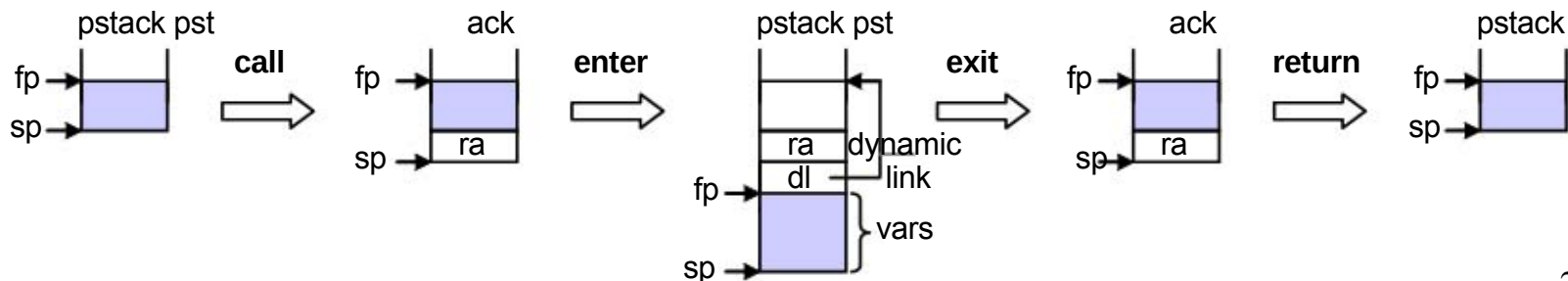
	<i>code</i>	<i>bytes</i>	<i>stack</i>
if (x > y) ...	load0 1		x
	load1	1	x y
	jle ...	3	-

Skup instrukcija μJVM

Poziv metoda

call s		<u>Call method</u> PUSH(pc+3); pc = s;
enter b1, b2		<u>Enter method</u> pars = b1; vars = b2; // in words PUSH(fp); fp = sp; sp = sp + vars; initialize frame to 0; for (i=pars-1; i>=0; i--) local[i] = pop();
exit		<u>Exit method</u> sp = fp; fp = POP();
return		<u>Return</u> pc = POP();

PUSH i POP
rade nad *pstack-om*



Poziv statičke funkcije

Primer

`c = m(a, b);`

load a

load b

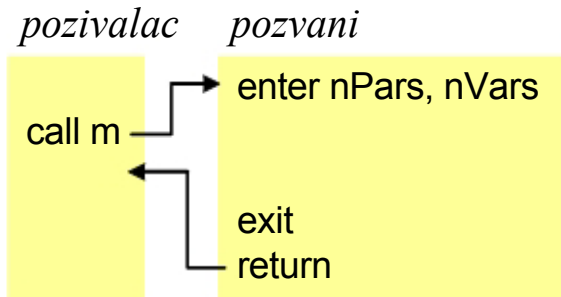
call m

store c

parameters are passed on the *estack*

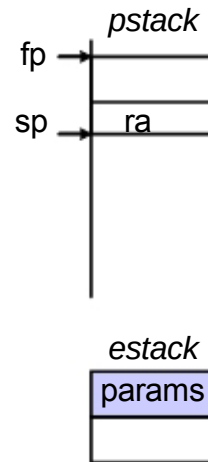
function value is returned on the *estack*

Okviri na programskom steku



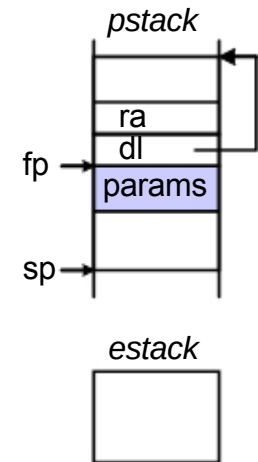
enter ... kreira okvir na psteku
exit ... uklanja okvir sa psteka

Enter instrukcija

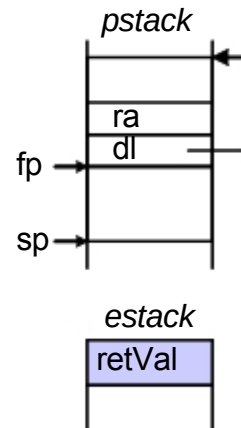


enter nPars, nVars

PUSH(fp); // dynamic link
 fp = sp;
 sp = sp + nVars;
 initialize frame to 0;
 for (i=nPars; i>=0; i--)
 local[i] = pop();

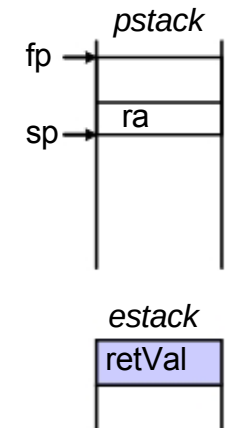


Exit instrukcija



exit

sp = fp;
 fp = POP();



Skup instrukcija μJVM

Ulaz/izlaz

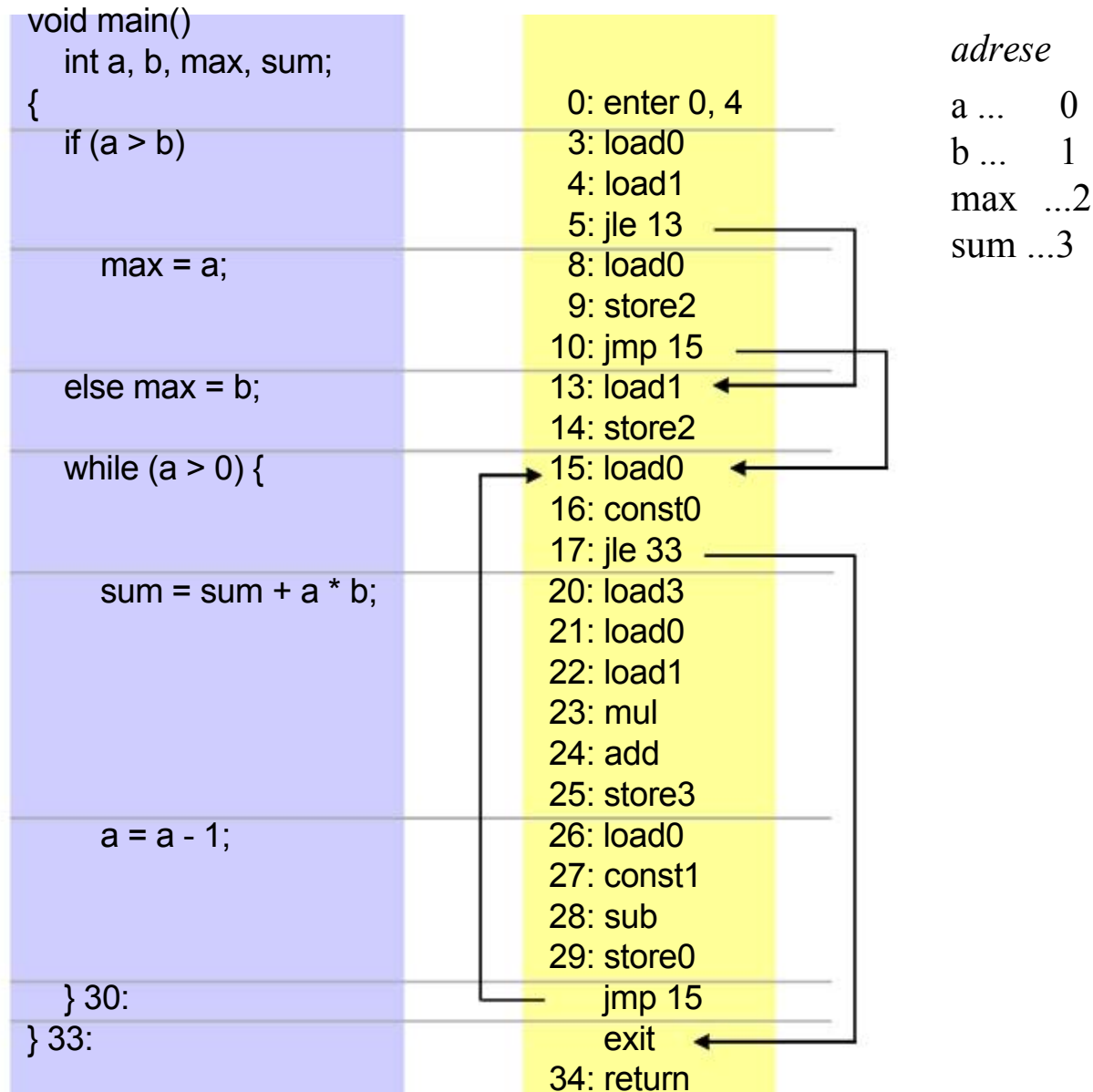
read	..., val	<u>Read</u> x = readInt(); push(x);
print	..., val, width	<u>Print</u> w = pop(); writeInt(pop(), w);
bread	..., val	<u>Read byte</u> ch = readChar(); push(ch);
bprint	..., val, width	<u>Print</u> w = pop(); writeChar(pop(), w);

ulaz sa System.in
izlaz na System.out

Ostalo

trap	b	<u>Throw exception</u> print error message b; stop execution;
-------------	---	---

Primer prevođenja statickog metoda



Format objektnog fajla

Sadržaj objektnog fajla u Mikrojavi

- informacije za punioca
 - veličina koda (u bajtovima)
 - veličina globalnih podatak (u rečima)
 - adresa *main* metoda
- programski kod

0	"MJ "
2	codeSize
6	dataSize
10	mainPc
14	code

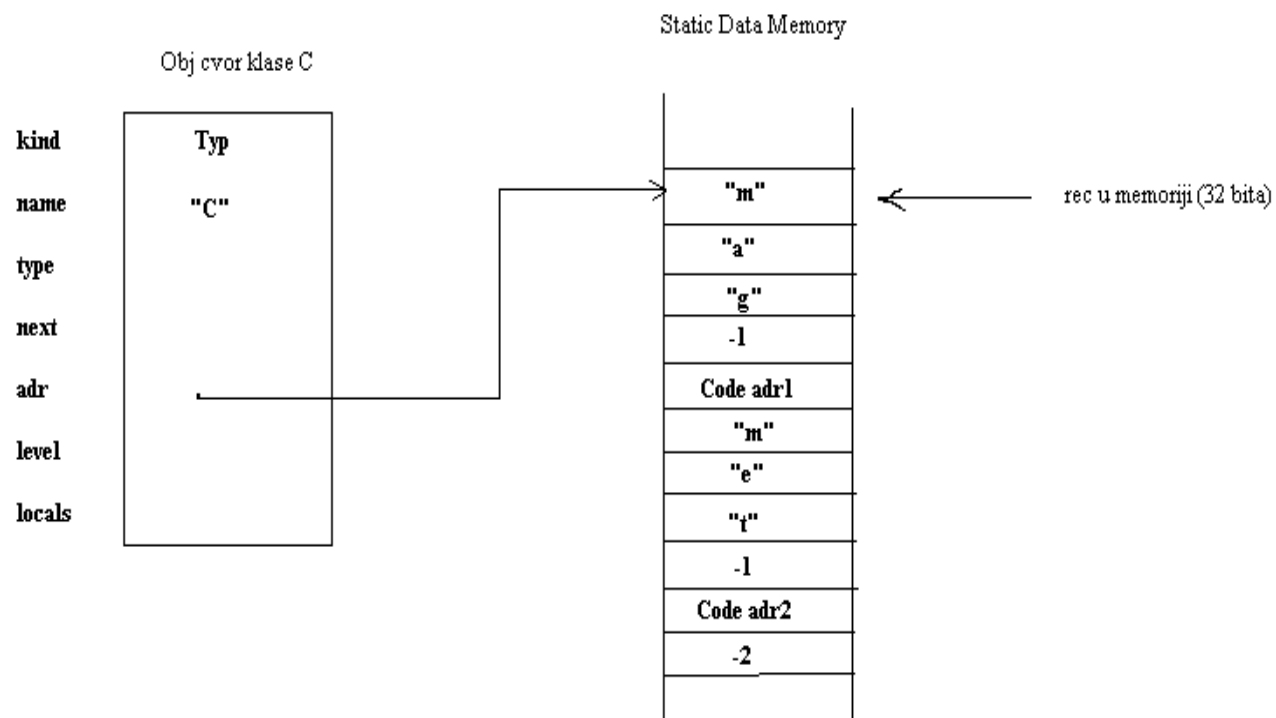
POZIV DINAMIČKIH METODA

- virtuelna tabela metoda

Svaka klasa, osim glavne koja predstavlja program, ima virtuelnu tabelu metoda (v-table). V-tabela se čuva u Static Data memoriji. Adresa prve reči v-tabele u Static Data memoriji se pamti u Obj čvoru klase, u tabeli simbola, u polju adr. Neka je, npr. data sledeća unutrašnja klasa:

```
class C {  
    int x,y, z;  
    int mag() {  
        return 3;  
    }  
    void met(int q) {  
        q=q+1;  
    }  
}
```

Struktura njene v-tabele je data na slici 1. Code adr1 je adresa početka koda u Code Memory koji odgovara metodi mag, slično, Code adr2 je adresa početka koda u Code Memory koji odgovara metodi met. U ovom primeru v-tabela zauzima 11 reči u Static Data. Broj -2 označava kraj tabele, a -1 razdvaja imena metoda.



- pravljenje novih instanci klase

Nove instance klase se prave korišćenjem operatora new. Novi objekat se pravi na Heap-u. Polja klase se numerišu od 1, jer je predviđeno da nulto polje sadrži adresu v-tabele svoje klase.

- pozivanje dinamičkih metoda

Neka je data sledeća sekvenca programskog koda:

```
C c=new C;      // klasa C definisana na strani 1
int a;
c.met(a);
```

Ovoj sekvenci odgovara sledeća sekvenca asemblerskih instrukcija MikroJava virtuelne mašine:

```
new 16  /* klasa C ima tri polja, plus jos jedno za adresu v-tabele, znaci 4 puta 4 bajta */
putstatic 0 /* pretpostavka je da je c statička promenljiva glavne klase na poziciji 0 */
getstatic 0
const 32  /* pretpostavka je da v-tabela za C počinje na adresi 32 u Static Data */
putfield 0 /* 32 je ubačeno u nulto polje objekta kojem pristupa referenca c */
/* poziv metode met */
getstatic 0 /* na stek izraza se stavlja c koje poziva metodu, pa prema tome predstavlja prvi stvarni
argument, tj. this */
getstatic 1 /* stavljanje drugog stvarnog argumenta. Pretpostavka je da je a statička promenljiva
glavne klase na poziciji 1 */
getstatic 0 // na stek se
getfield 0 // stavlja adresa v-tabele klase objekta na koji ukazuje referenca c
invokevirtual 'm' 'e' 't' -1 /* poziv metode met. -1 označava kraj instrukcije */
```

Instrukcija virtuelne mašine, **invokevirtual** *imeMetode*, prvo uzme sa steka izraza adresu v-tabele, pristupi joj i poređenjem karaktera traži da li se u njoj nalazi ime metode koje se poklapa sa imenom koje je operand instrukcije. Ako je ime nađeno, iz v-tabele se čita adresa početka koda metode, koja se nakon stavljanja trenutne vrednosti PC-ja na stek procedura upisuje u PC. Time se prelazi na izvršenje koda odgovarajuće metode.

Promenljivoj *c* može biti dodeljen objekat tipa klase *C*, ali i tipa bilo koje potklase klase *C*. Znači, da se u doba prevođenja ne zna kog će dinamičkog tipa *c* zaista biti. Ali će se u nultom polju objekta koji *c* referencira nalaziti adresa v-tabele odgovarajućeg dinamičkog tipa, tj. klase. Time je obezbeđeno da bude pozvana ispravna metoda u zavisnosti od tipa objekta, a ne tipa reference (polimorfno ponašanje).