Java összetett példa

Készítette: Goldschmidt Balázs, BME IIT, 2023.

A feladatban egy veremszámítógépet készítünk. A megoldás alapját ismét egy parancsfeldolgozó mag fogja alkotni, de most a parancsokat objektumorientáltan kell elkészíteni, és a számítások értékeinek tárolásához egy vermet kell használni. A számológép egész számokon végzett műveleteket és ezekből álló programokat tud majd végrehajtani.

A veremszámológép lényege, hogy a számok, amiken műveleteket akarunk végezni, a vermen helyezkednek el. A műveletek mindig a verem tetején lévő számokat használják, és az eredmények is ide kerülnek. A felhasznált számok kikerülnek a veremből.

Pl. a **2*(3+5)** *kifejezést a következő lépésekben hajtjuk végre és értékeljük ki (a doboz jelképezi a vermet):*

1. *Betesszük 2-t*	***2.*** *Betesszük 3-at*	***3.*** *Betesszük 5-t*	***4.*** *Összeadunk (+)*	***5.*** *Összeszorzunk (*)*
		5		
	3	*3*	*8*	
2	*2*	*2*	*2*	*16*

Összesítve, az egyes műveletek egymás után:

push 2
push 3
push 5
összeadás (+)
szorzás (*)

Egy letisztultabb, másik jelöléssel (ezt „fordított lengyel jelölés”-nek nevezik): ***2 3 4 + ****.*

Ezzel a módszerrel tetszőleges kifejezés leírható és kiszámolható, még a zárójelezéssel sem kell bajlódni.

1. A veremszámítógép alapjai

a) Készítsük el a főprogramot, amely a korábban már begyakorolt módon fog parancsokat olvasni a szabványos bemenetről.
b) Legyen egy statikus attribútumunk a veremhez *stack* néven, típusként használjuk a *Deque<Integer>* interfészt megvalósító *ArrayDeque<Integer> osztályt*. Az alábbi metódusokat használhatjuk majd a feladat megoldásához:
   - **void push(Integer d)**: elhelyez egy Integer-t a verem tetején.
   - **Integer pop()**: leveszi és visszaadja a verem tetején levő elemet
   - **Integer peek()**: a helyén hagyja, de visszaadja a verem tetején levő elemet
   - **Iterator<Integer> iterator()**: iterálást biztosít a verem elemein.
   - **int size()**: visszaadja a veremben levő elemek számát
   - **boolean isEmpty()**: megadja, hogy a verem üres-e.

2. Az alapparancsok kezelése

Valósítsuk meg a parancsokat. Az eddigiektől eltérően most objektumorientált megoldást alkalmazunk. A függvényeket különböző osztályok egy-egy metódusa helyettesíti majd.

a) Készítsünk egy interfészt (*Command*), amelyiknek egy metódusa van:
   void execute(String[] cmd)
   A paramétere a szokásos string tömb a paranccsal és argumentumaival.
b) Készítsünk el az alábbi osztályokat a megadott *execute* implementációkkal:

Osztály neve	Execute metódus specifikációja
Exit	Meghívja a *System.exit(0)* metódust.
List	Kiírja a verem tartalmát a szabványos kimenetre. Ne feledjük, hogy a verem a *Main.stack* változónévvel érhető el.
Push	A *cmd[1]* egész értékét a veremre teszi.
Pop	Leveszi a verem tetejéről a legfelső elemet.
Dup	A verem tetején levő elemet megduplázza. Ha a verem tartalma az alábbi (jobbra a később berakott): [3;2;1], akkor a parancs után a verem tartalma ez lesz: [3;2;1;1]
Read	Beolvas egy egész számot a szabványos bemenetről és a verem tetejére teszi. *Tipp: a beolvasáshoz használjuk ugyanazt a Scanner objektumot, amivel a parancsokat is olvassuk!*
Write	A verem tetején levő számot leveszi és kiírja a szabványos kimenetre.

Vegyük észre, hogy az eddig alkalmazott függvény-alapú megoldásunkkal szemben most a függvények helyett valósítjuk meg az osztályokat. Az egyes függvények törzse kerül a megfelelő osztály execute metódusába.

c) Készítsünk a főprogramban (a *main* metódus elején) egy inicializáló kódrészletet, ahol
   a) létrehozunk egy *HashMap<String, Command>* tároló objektumot (*map*) a parancsok tárolásához
   b) létrehozunk minden osztályból egy példányt, pl.:
      Command c = new Exit();
   c) a példányokat hozzáadjuk az osztály kisbetűs nevével a HashMap-hez, pl.:
      map.put("exit", c);

Vegyük észre, hogy a két fenti sort össze is vonhatjuk, pl.:
map.put("exit", new Exit());

d) A parancsfeldolgozó ciklusunk ne a megszokott *if-else-if...* listából álljon, hanem a fenti *map*-et használja, valahogy így:

```
String line = sc.nextLine();
String[] cmd = line.split("␣");  // szóközt tartalmazó string
Command c = map.get(cmd[0]);
c.execute(cmd);
```

Vegyük észre, hogy a fenti megoldással kényelmesen, a parancsbeolvasó rutin módosítása nélkül tudjuk új paranccsal bővíteni a rendszerünket. Ehhez csak az kell, hogy az új parancsot megvalósító osztályt példányosítsuk és hozzáadjuk a megfelelő névvel a map*-hez.*

3. Matematikai műveletek megvalósítása

a) Készítsünk el az alábbi osztályokat a megadott *execute* implementációkkal:

Osztály neve	Execute metódus specifikációja
Add	A verem tetején levő 2 számot leveszi a veremről és az összegüket teszi a verem tetejére. Pl. ha a verem tartalma (2 az utoljára betett érték): [4;3;2], akkor a művelet után ez lesz: [4;5].
Sub	A verem tetején levő 2 számot leveszi a veremről és a különbségüket teszi a verem tetejére. Pl. ha a verem tartalma ez: [4;3;2], akkor a művelet után ez lesz: [4;-1].
Mult	A verem tetején levő 2 számot leveszi a veremről és a szorzatukat teszi a verem tetejére. Pl. ha a verem tartalma ez: [4;3;2], akkor a művelet után ez lesz: [4;6].
Div	A verem tetején levő 2 számot leveszi a veremről és a hányadosukat teszi a verem tetejére. Pl. ha a verem tartalma ez: [4;2;6], akkor a művelet után ez lesz: [4;3].

b) Az osztályokat az előző feladatban látott módon példányosítsuk és adjuk hozzá a map-hez, majd az első oldalon mutatott példával próbáljuk ki, hogy működnek-e!

Ezzel megvalósítottunk egy egyszerű veremszámológépet.

4. „Programozhatóság” megvalósítása

Az előző feladatok megoldásával kész a veremszámológépünk. Hogy ebből programozható veremszámítógép váljon, néhány speciális parancsra és a parancsok fájlból való olvasására lesz szükség.

Most ott tartunk, hogy pl. a következő „programot” végre tudja hajtatni az alkalmazásunk (beolvas két számot és kiírja az összegüket):

```
read
read
add
write
```

A gond csak az, hogy ezeket a parancsokat mindig be kell gépelni. Jó lenne a parancsokat fájlból olvasni. Ezt fogjuk most megtenni.

a) Az alkalmazásunk induláskor nézze meg, hogy kapott-e parancssori argumentumot (a *main* metódus *args* tömbje nagyobb-e, mint nulla)
b) Ha a tömb mérete nulla, akkor minden történjen úgy, mint eddig.
c) Ha a tömb mérete legalább egy, akkor
    i. az első elemet (*args[0]*) egy fájl neveként használva nyissuk meg olvasásra a fájlt, és soronként olvassuk be. Az egyes sorokat tegyük bele egy *LinkedList<String>* típusú listába (*lines*))!
    ii. Vegyünk fel utasításszámláló céljából egy egészértékű, statikus attribútumot (*pc*), és az értékét állítsuk 0-ra!
    iii. Egy ciklusban vegyük ki a *pc*-nek megfelelő indexű elemet a *lines* listából, és az ismert módon dolgozzuk fel a kapott stringet! Ha túlcímeznénk, akkor a program érjen véget.
    iv. Eggyel növeljük meg a *pc* értékét! Így a ciklus szépen végig tud haladni a beolvasott és a *lines* nevű listában tárolt parancsokon.
d) Próbáljuk ki az alkalmazást a fenti 4 soros programmal! A parancssori argumentumokat a projekt nevén jobbklikkel feljövő menüben a *Run As / Run Configurations* menüponttal állíthatjuk be. A felbukkanó ablakban az *Arguments* fület választva tudjuk megadni az argumentum értékét (a fájl nevét).
   A programocskénkat tartalmazó fájlt magát pedig legkönnyebben úgy hozhatjuk létre, ha a projekt nevén jobbklikkelve kiválasztjuk a *New/File* menüpontot, majd megadjuk a szükséges fájlnevet. Ekkor a fájl megnyílik szerkesztésre az eclipse-ben. Ha beleírtuk a parancsokat, mentsük el, majd futtassuk az alkalmazást!

5. Elágazó és ugró utasítások megvalósítása

Egy program nem program, ha csak sorban képes utasításokat végrehajtani. Bővítsük hát elágazással (az if *megfelelőjével) az eddigi alkalmazásunkat! A legkönnyebben ezt úgy tudjuk megtenni, ha a verem tetején levő értéktől függően tudjuk különböző helyeken folytatni a végrehajtást.*

a) Vezessük be a címkéket! Amelyik sor „#”-kal kezdődik, címkének fog számítani, amire később az ugró és elágazó utasítások hivatkozni tudnak. Ehhez vegyünk fel egy új *HashMap<String,Integer>* típusú, statikus attribútumot (*labels*)! Ne felejtsük el inicializálni!

b) Ha a fájlból olvasáskor „#”-kal kezdődő sort olvasunk, akkor azt nem tároljuk le, hanem csak a következő sor sorszámát (*lines* változó *size()* metódusának visszatérési értékét) tároljuk el a „#” kezdetű sorral, mint kulccsal a *labels* map-ben. Így, ha később egy utasítás erre hivatkozik, akkor a *labels*-ből kikereshető, hogy melyik sorra is utal.

c) Vezessük be a következő elágazó illetve ugró utasításokat:

Osztály neve	Execute metódus specifikációja
Jump	Az argumentumban megadott címkéhez tartozó értéknél eggyel kisebbre állítja a *pc* értékét (mert a végrehajtó ciklus mindig növel egyet a *pc*-n).
OnZero	Leveszi a verem tetejéről az ott levő számot. Ha ez 0 volt, akkor az argumentumban megadott címkéhez tartozó értéknél eggyel kisebbre állítja a *pc* értékét.
OnNonZero	Leveszi a verem tetejéről az ott levő számot. Ha ez nem 0 volt, akkor az argumentumban megadott címkéhez tartozó értéknél eggyel kisebbre állítja a *pc* értékét.
OnNegative	Leveszi a verem tetejéről az ott levő számot. Ha kisebb, mint 0, akkor az argumentumban megadott címkéhez tartozó értéknél eggyel kisebbre állítja a *pc* értékét.

d) Próbáljuk ki az alkalmazásunkat! Írjunk rövid programocskát, amely kiírja a számokat 1-től 10-ig!
Pl.:

```
push 0
#ciklus
push 1
add
dup
write
dup
push 10
sub
onnonzero #ciklus
exit
```

6. Változók használata

Bővítsük a veremszámítógépünket változók használatával! Ehhez kelleni fog egy olyan tároló, amely változónevekhez értéket tud nyilvántartani. Legyen ez egy *HashMap<String,Integer>* típusú, *vars* nevű objektum. A használata egyszerű. Két műveletre van csak szükségünk: *store* és *load*. A *store* a megadott nevű változóba eltárolja a verem tetején levő értéket. A *load* az adott nevű változó értékét a verem tetejére helyezi.

Pl1: a következő program beolvas 2 számot és a szorzatukat letárolja az „x” nevű változóban:

```
read
read
mult
store x
```

Pl2: Az alábbi program pedig beolvas 10 számot és kiírja a legnagyobbat (indentálva az olvashatóság kedvéért):

```
read
store max
push 1
#ciklus
  read
  dup
  load max
  sub
  push -1
  mult
  onnegative #kisebb
    dup
    store max
  #kisebb
  pop
  push 1
  add
  dup
  push 10
  sub
onnonzero #ciklus
load max
write
```

7. Extra feladat: függvények definiálása és hívása

Bővítsük a veremszámítógépünket függvények definiálásával!

a) Ehhez szükségünk lesz egy újabb *Deque<Integer>* típusú veremre (*frame*), amire a függvényhíváskor érvényes *pc* érték kerül, hogy a függvény visszatérésekor ott folytassuk, ahonnan a függvényt meghívtuk.

b) Kell egy új jelölés a függvényekhez, ez a címkékhez hasonlóan legyen *@függvénynév* alakú, és a címkékhez hasonlóan beolvasáskor egy *HashMap<String, Integer>* típusú változóban (*functions*) tároljuk, hogy milyen nevű függvény melyik soron kezdődik.

c) Kell két új utasítás: *call* és *return*.
A *call* a paraméterében megadott függvénynél folytatja a futást (hasonlóan a *jump*-hoz), de előtte beteszi az aktuális *pc* értéket a *frame*-be.
A *return* a frame tetején levő számot leveszi, és a pc-nek értékül adja.
Fontos, hogy a függvényeink a paramétereiket a közös stack-en tudják átvenni, és ami a stack-en marad, azt a visszatéréskor láthatja a hívó. A memória viszont közös.

d) Extra speciális szorgalmi feladat: a memóriakezelés legyen olyan, hogy a függvényeknek legyen lokális memóriája is, ami a store *és* load *parancsok mintájára működik (*put *és* get*), de a függvényhíváskor jön létre, és a visszatéréskor eldobjuk. A legjobb, ha egy verem működésű tárolóba (*Deque<HashMap<String, Integer>> memstack*) tesszük az éppen futó függvény memóriáját megvalósító* HashMap<String, Integer>*-t. Ezt* HashMap*-et a* call *hozza létre és teszi a* memstack*-re, és a* return *veszi le onnan. A* put *és a* get *pedig mindig a memstack tetején levő* HashMap*-et használja.*

A fentiek alapján például egy négyzetre-emelő függvény és meghívása így nézhet ki. Beolvasunk egy számot, meghívjuk a négyzetre-emelést, kiírjuk az eredményt és kilépünk. A négyzetre-emelés pedig a verem tetején levő számot duplikálja, meghívja a szorzást a két példányon, és végül visszatér.

```
read
call @square
write
exit
@square
  dup
  mult
  return
```