Александар Картељ
Владимир Филиповић
Душан Тошић

ОБЈЕКТНО ОРИЈЕНТИСАНО ПРОГРАМИРАЊЕ

ПРОГРАМСКИ ЈЕЗИК ЈАВА – 1

Математички факултет
Универзитет у Београду
2023. година

Универзитет у Београду

Математички факултет

Објектно оријентисано програмирање

Програмски језик Јава – 1

Александар Картељ, Владимир Филиповић, Душан Тошић

Београд, 2023.

Садржај

Предговор

Ова књига је, првенствено, намењена студентима Математичког факултета у Београду и требало би да послужи као уџбеник за предмет Објектно оријентисано програмирање. Међутим, књига може бити од користи свакоме ко жели да научи програмски језик Јава и да се упозна са принципима програмирања. У књизи је покривена верзија Јава 17LTS.

На српском говорном подручју постоји велики број књига које се односе на програмски језик Јава и пратеће технологије. Ова књига је специфична због повезаности са предметом Објектно оријентисано програмирање и усклађена је са програмом овог предмета. Књига представља свеобухватни водич за програмски језик Јава, али не и за све његове пратеће библиотеке. Опис коришћења свих кључних библотека језика Јава превазилази обим ове књиге. За боље разумевање пожељно је да читалац познаје програмски језик C, али није неопходно.

За опис конструкција Јава језика коришћена је Бекусова нотација. Скоро сваки формални опис синтаксе Јава-конструкција пропраћен је примерима. Додатно, већина поглавља садржи већи број комплетно урађених задатака. Кроз решења задатака приказују се најважније могућности језика Јава и принципи програмирања. За највећи број решења наведени су тест примери да би се видело како је организован улаз и у каквом облику се добијају излазни подаци. Решени примери из уџбеника доступни су и на GitHub страни посвећеној уџбенику https://github.com/matf-oop-java/tom-1.

Књига се састоји од 15 поглавља, два додатка и литературе. На крају сваког поглавља су наведена питања и задаци за вежбање. Ако неко учи било који програмски језик, па самим тим и програмски језик Јава, пожељно је да самостално уради што већи број задатака за вежбање.

Аутори су настојали да свуда, где је то могуће, користе ћирилично писмо. Коментари у програмима и излазни резултати, који садрже текст, исписани су ћирилицом. Када су у питању имена програмских технологија, она су скоро искључиво писана у оригиналу, тј. без транскрипције са енглеског језика. Изузетак су назив програмског језика Јава, реч квиксорт (енг. quick sort) и још неколико других речи. Оригинални запис речи Јава је задржан у ситуацијама када реч учествује

у формирању назива неке специфичне Јава технологије, на пример, Java Security API или Java FX.

С обзиром на то да је ово прво издање књиге, грешке су могуће. Штампарске грешке у рукопису су скоро неизбежне, иако су текст пажљиво прочитали аутори и већи број колега. Међу њима велику захвалност дугујемо др Милани Грбић, која је, поред пажљивог читања, помогла и у конципирању питања и задатака на крајевима поглавља и др Сташи Вујичић-Станковић, која је пажљиво прочитала сва поглавља књиге и дала врло детаљне и корисне коментаре. Поред њих, захваљујемо се проф. др Драгану Матићу на уочавању неких терминолошких и суштинских грешака, проф. др Филипу Марићу на предложеној реорганизацији примера у почетним поглављима књиге и проф. др Предрагу Јаничићу на давању смерница за припрему књиге у складу са правилима издавача. Захваљујемо се и нашим студентима који су помогли у отклањању словних грешака и логичких недоследности. Реч је о студентима: Андреа Андрејевић, Ирина Шевић, Стефан Миленковић, Лука Петковић и Урош Динић. Унапред се извињавамо онима које смо евентуално заборавили да споменемо.

Посебну захвалност дугујемо рецензентима проф. др Ненаду Митићу и проф. др Вељку Милутиновићу. Они су пажљиво прочитали цео рукопис и корисним сугестијама допринели да ова књига буде квалитетнија.

Аутори

6. Коришћење класа и објеката испоручених уз JDK

У овом поглављу ће бити представљене неке од основних могућности Јава библиотеке класа попут исписа и читања са стандардног излаза/улаза, мерења протеклог времена, рада са текстом, псеудослучајним бројевима, математичким функцијама итд.

6.1. Приступ систему, класа System

Класа `System` се може назвати услужном класом (енг. utility class) будући да се не очекује прављење њених објеката. Акценат је на позивању њених метода у тзв. статичкој нотацији, која је коришћена и у претходним поглављима. Статичка поља и методи ће бити детаљније разматрани у секцијама 8.3.3 и 8.4.4. Засад је довољно рећи да су у питању ентитети који нису зависни од постојања објекта посматране класе, тј. може им се приступати коришћењем имена класе на следећи начин (што доста подсећа на стил процедуралног програмирања):

```
NazivKlase.statickiMetod()
NazivKlase.statickoPolje
```

6.1.1. Приказ текста

Класа `System` има статичко поље `System.out`, које представља објекат везан за стандардни излаз (попут `stdout` у језику C). `System.out` подразумевано показује на конзолу па се запис текста манифестује исписом на конзоли.

Овај објекат поседује методе (нестатичке) којима се приступа употребом тачка нотације. Следи преглед неких, често коришћених, метода.

```
System.out.print(tip arg)
System.out.println(tip arg)
System.out.printf(String format, Object...args)
```

`System.out.print()` исписује аргумент на стандардни излаз (аргумент је произвољног типа, било примитивног или објектног).

`System.out.println()` ради исто што и `System.out.print()` и уз то додаје ознаку за прелазак у нови ред.

`System.out.printf()` исписује текст форматирано, попут функција за форматирани испис у C-у (`printf()`, `sprintf()`, `fprintf()`).

Пример Написати Јава програм који приказује употребу форматираних и неформатираних исписа различитих типова података на стандардном излазу.

```
public class RazlicitiIspisi {
    public static void main(String[] args) {
        System.out.print("Пример текста");
        System.out.print(" са конкатенацијом и бројем "+4);
        System.out.println(); // испис празног реда
        System.out.println(67.4);
        double x = 26.43462;
        int y = 43243;
        float z = 1645.14f;
        System.out.printf("x=%9.6f y=%8d z=%.3f", x, y, z);
        System.out.println();
        String s = "Неки текст";
        char c = 'c';
        System.out.printf("Стринг се умеће форматом %%s %s", s);
        System.out.printf(" док се карактер умеће форматом %%%c%s", c,
                                                    System.lineSeparator());
        System.out.printf("%s\t%s%n", "KPAJ", "ISPISA");
    }
}
```

Када се користи форматирани испис, први број после знака `%` представља број места која ће се користити за комплетан запис броја. Ако нема овог броја, број места се аутоматски прилагођава дужини записа броја. Број после тачке означава колико места се користи за разломљени део броја. На пример, `%9.6f` значи да ће бити употребљено девет места, од чега ће 6 бити употребљено за запис разломљеног дела реалног броја.

Будући да знак `%` има специјално значење код форматираног исписа, да би се записао знак `%` потребно је записати га два пута у оквиру ниске за запис формата (`%%`).

`System.lineSeparator()` враћа секвенцу за крај реда прилагођену оперативном систему, али се ово може краће записати као `%n` (у оквиру форматираног исписа).

Приметити да је приликом исписа могуће комбиновати различита писма – ово је последица тога што су ниске у Јави конципиране као секвенце Unicode карактера.

Следи резултат извршавања програма.

```
Пример текста са конкатенацијом и бројем 4
67.4
x=26.434620 y=   43243 z=1645.140
Стринг се умеће форматом %s Неки текст док се карактер умеће форматом %c
KPAJ    ISPISA
```

Док је `System.out` надлежан за испис информација у току нормалног функционисања програма, `System.err` је препоручени начин за испис грешака које се јављају у току извршавања или приликом покретања програма. На пример, ако би се тражило да корисник унесе број, а корисник уместо тога унесе секвенцу карактера „4543-АФД-325-11“, програм би могао да испише на `System.err` упозорење о лошем запису броја и да прекине извршавање или да захтева поновни унос. Списак доступних метода у оквиру `System.err` је идентичан као и у случају `System.out`, што је и очекивано с обзиром да оба представљају ток података под називом `PrintStream` (више о токовима у секцији 15.1).

Локација исписа `System.out` и `System.err` се може променити применом метода `System.setOut()` и `System.setErr()`. На пример, може се направити датотека која ће се користити за испис, уместо исписа на конзолу.

6.1.2. Мерење протеклог времена

Мерење времена, у оквиру програма, може бити корисно у анализи перформанси захтевних делова кода. `System` класа омогућава два начина за мерење времена:

1. употребом метода `System.currentTimeMillis()` и
2. употребом метода `System.nanoTime()`.

Принцип рада оба метода је исти – враћају време протекло од неког референтног тренутка у прошлости (Unix epoch - 1. јануар 1970 по Гриничу), с тим што је други метод прецизнији, јер рачуна протекло време у наносекундама, за разлику од првог који рачуна у милисекундама. На овај начин је могуће индиректно закључити колико траје неки интервал тако што се одузму „апсолутна“ времена догађаја за крај и почетак интервала.

Пример 2. Написати Јава програм који мери време извршавања метода за сабирање бројева од 1 до n. Тестирати програм за различите вредности n, као и за различите методе за мерење времена. Упоредити измерене вредности у милисекундама. δ

```
public class IzmeriVreme {
    public static long sumiraj(int n) {
        long suma = 0;
        for (int i = 0; i < n; i++)
               suma += i;
        return suma;
    }

    public static void main(String args[]) {
        for (int n = 10000000; n <= 1000000000; n *= 10) {
               long pocetak1 = System.nanoTime();
               long suma = sumiraj(n);
               System.out.printf("Сума бројева до %d је %d%n", n, suma);
               long kraj1 = System.nanoTime();
               System.out.println("Време у ns са nanoTime(): "
                                                  + (kraj1 - pocetak1));
               System.out.printf("Време у ms са nanoTime(): %.5g%s",
              (kraj1-pocetak1)/1000000.0,System.lineSeparator());

               long pocetak2 = System.currentTimeMillis();
               suma = sumiraj(n);
               System.out.printf("Сума бројева до %d је %d%n", n, suma);
               long kraj2 = System.currentTimeMillis();
               System.out.println("Време у ms са currentTimeMillis(): "
                                                  + (kraj2 - pocetak2));
              System.out.println("-----------------------------------");
        }
    }
}
```

Може се приметити да `nanoTime()` даје много прецизнија мерења и да број протеклих милисекунди метода `currentTimeMillis()` постаје тачнији (или мање погрешан) са растом n, тј. дужином трајања метода `sumiraj()`.

Следи резултат рада програма.

```
Сума бројева до 10000000 је 49999995000000
Време у ns са nanoTime(): 10856500
Време у ms а nanoTime(): 10.857
Сума бројева до 10000000 је 49999995000000
Време у ms са currentTimeMillis(): 4
-----------------------------------------------
Сума бројева до 100000000 је 4999999950000000
Време у ns са nanoTime(): 22943200
Време у ms са nanoTime(): 22.943
Сума бројева до 100000000 је 4999999950000000
Време у ms са currentTimeMillis(): 23
-----------------------------------------------
Сума бројева до 1000000000 је 499999999500000000
Време у ns са nanoTime(): 228048000
Време у ms са nanoTime(): 228.05
Сума бројева до 1000000000 је 499999999500000000
Време у ms са currentTimeMillis(): 228
-----------------------------------------------
```

6.1.3. Захтев за покретањем скупљача отпадака, метод System.gc()

Као што је раније речено, Јава аутоматски уклања са хипа податке који више нису у употреби, тј. на које се више не референцира. Скупљач отпадака се активира по потреби и његов рад се заснива на коришћењу разних хеуристика. Захваљујући хеуристикама, скупљач отпадака се прилагођава конкретном процесу, предвиђа употребу меморије и слично. Квалитет скупљача отпадака може се оценити коришћењем два критеријума:

1. колико је, у просеку, неки објекат „чекао“ у меморији од момента када је постао непотребан до момента уклањања и

2. колики ефекат на перформансе целе виртуелне машине има рад скупљача отпадака.

Ова два критеријума су често у колизији, тј. минимизација времена чекања на уклањање подразумева чешће активације скупљача отпадака и самим тим смањење перформанси виртуелне машине.

(Напомена: добро написан C програм, у којем се меморија експлицитно ослобађа чим више није неопходна (наредбом `free()`), нуди максималне перформансе уз минимизацију ефекта „чекања“.)

Програмер, дакле, у Јави не може уклањати податке из меморије самостално. Највише што може је да сугерише скупљачу отпадака да се активира преко метода `System.gc()`. Ретке су ситуације у којима програмер може да има користи од експлицитног позивања овог метода и генерално се његово позивање не препоручује.

Пример 3. Написати Јава програм који користи једну референцну променљиву за прављење великог броја нових ниски на хипу. Ово за последицу има велики број нереферисаних објеката током рада програма. Упоредити количину доступне и слободне меморије на хипу у варијантама:

1. када скупљач сам одређује кад ће се активирати и
2. када му корисник повремено сугерише скупљање помоћу `System.gc()` метода. δ

```
public class PozoviGC
    static void stanjeMemorije() {
        long slobodno = Runtime.getRuntime().freeMemory()/1024/1024;
        long maksimalno = Runtime.getRuntime().totalMemory()/1024/1024;
        System.out.printf("%dMB од %dMB | ", slobodno, maksimalno);
    }

    public static void main(String[] args) {
        int n = 10000000;
        int nIspis = 200000;
        String s;
        long pocBezGC = System.nanoTime();
        System.out.println(
          "Величине слободне меморије без позивања gc() пред испис");
        for (int i = 0; i < n; i++) {
               s = new String("Текст под редним бројем " + i);
               if (i % nIspis == 0)
                      stanjeMemorije();
```

```
        }
        System.out.printf("%nВреме без GC\t%.2f%n",
                                  (System.nanoTime() - pocBezGC) / 1e6);

        long pocSaGC = System.nanoTime();
        System.out.println(
          "Величине слободне меморије са позивањем gc() пред испис");
        for (int i = 0; i < n; i++) {
               s = new String("Текст под редним бројем " + i);
               if (i % nIspis == 0) {
                      System.gc();
                      stanjeMemorije();
               }
        }
        System.out.printf("%nВреме са GC\t%.2f%n",
                                   (System.nanoTime() - pocSaGC) / 1e6);
    }
}
```

У варијанти без употребе `System.gc()` метода види се да програм ради са већом (флексибилнијом) количином меморије, што омогућава да скупљач не мора често да се активира. Тек повремено, приликом значајног смањења удела слободне меморије, скупљач се самостално активира и обрише непотребне податке. Са друге стране, у режиму експлицитног позивања скупљача отпадака, количина слободне меморије је врло близу максимуму. Време извршавања је у другом случају нарушено, што је очекивана последица скупог процеса скупљања отпадака.

Следи резултат рада програма.

```
Величине слободне меморије без позивања gc() пред испис
1021MB од 1024MB | 978MB од 1024MB | 987MB од 1024MB | 973MB од 1024MB |
957MB од 1024MB | 941MB од 1024MB | 927MB од 1024MB | 1015MB од 1024MB |
999MB од 1024MB | 983MB од 1024MB | 967MB од 1024MB | 955MB од 1024MB |
939MB од 1024MB | 923MB од 1024MB | 907MB од 1024MB | 891MB од 1024MB |
879MB од 1024MB | 863MB од 1024MB | 847MB од 1024MB | 831MB од 1024MB |
815MB од 1024MB | 799MB од 1024MB | 787MB од 1024MB | 771MB од 1024MB |
755MB од 1024MB | 739MB од 1024MB | 723MB од 1024MB | 711MB од 1024MB |
695MB од 1024MB | 679MB од 1024MB | 663MB од 1024MB | 647MB од 1024MB |
631MB од 1024MB | 619MB од 1024MB | 603MB од 1024MB | 587MB од 1024MB |
571MB од 1024MB | 555MB од 1024MB | 543MB од 1024MB | 527MB од 1024MB |
511MB од 1024MB | 495MB од 1024MB | 479MB од 1024MB | 463MB од 1024MB |
451MB од 1024MB | 435MB од 1024MB | 1019MB од 1024MB | 1003MB од 1024MB |
```

```
987MB од 1024MB | 975MB од 1024MB |
Време без GC    373.09
Величине слободне меморије са позивањем gc() пред испис
55MB од 56MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB
| 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од
32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB
од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB
| 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од
32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB
од 32MB | 31MB од 32MB | 55MB од 56MB | 31MB од 32MB | 31MB од 32MB |
31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB
| 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од 32MB | 31MB од
32MB | 31MB од 32MB | 31MB од 32MB |
Време са GC    445.92
```

6.1.4. Излазак из апликације, метод System.exit()

Попут функције `exit()` у програмском језику C и Јава има могућност да прекине извршавање актуелног програма. У Јави се, заправо, гаси читава Јава виртуелна машина. Метод је статички и налази се у класи `System`. Његово заглавље је:

```
public static void exit(int status)
```

Статус 0 указује на успешан завршетак извршавања док не-нула вредности указују на неуспешно извршавање, тј. на грешку.

Пример 4. Написати Јава програм који реализује и тестира метод за исписивање карактеристика монитора, при чему су аргументи метода ширина и висина. Потребно је израчунати број тачака (пиксела) и сврстати монитор у категорију стандардни или широки на основу односа ширине и висине: монитор је широк уколико је ширина два или више пута већа од висине. Такође је потребно на одговарајући начин реаговати на некоректне уносе. δ