

# Siniša Nikolić

# Sadržaj

- ☞ Rad sa Stack memorijom,
- ☞ Rad sa Heap memorijom i Objekti u javi,
- ☞ Rad sa nizovima

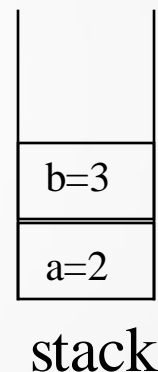
# Rad sa Stack memorijom – čuvanje promenljivih

- ☹ Promenljive se čuvaju u delu memorije označen kao *stack*.
- ☹ Stek je poseban deo memorije u kojem se smeštaju sve promenljive koje se koriste u programu.
- ☹ Princip funkcionisanja steka je takav da se promenljiva uvek dodaje na vrh steka (ređaju se jedna na drugu) i može se ukloniti sa steka samo ona koja se nalazi na vrhu (uklanjaju se u obrnutom redosledu od dodavanja) tj. LIFO (*Last-In-First-Out*) poredak.

# Rad sa Stack memorijom – čuvanje promenljivih


- ☕ Vrednost za promenljive primitivnih tipova se čuva na steku tj. na *stack*-u se zauzima (alocira) memorija u kojoj će se smestiti vrednost primitivnog tipa.

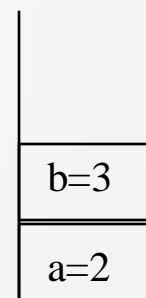
```
public static void main(String[]  
    args) {  
    int a = 2;  
    int b = 3; ←  
    int c = 5;  
}
```



# Rad sa Stack memorijom – čuvanje promenljivih

- ☞ Za svaku funkciju se njene promenljive tretiraju kao lokalne (traju koliko i sama funkcija).

```
public static int obradiZbir (int bF,  
    int aF) {  
    int rF = aF + bF;  
    return rF;  
}  
  
public static void main(String[] args) {  
    int a = 2;  
    int b = 3;   
    int c = obradiZbir(b,a);  
}
```

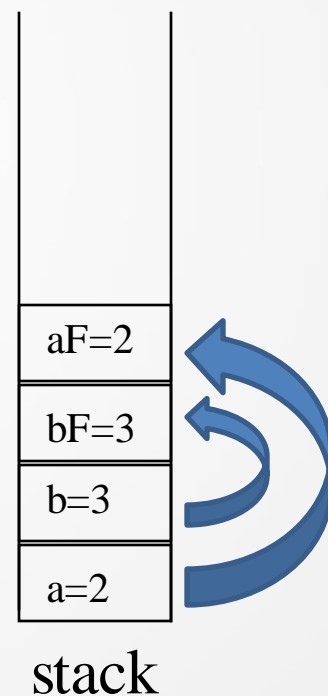


stack


# Rad sa Stack memorijom – čuvanje promenljivih

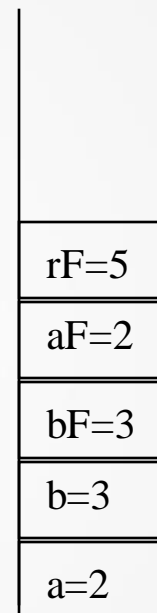
- ☕ Pri pozivu funkcija na stek se redom dodaju promenljive koje su ulazni parametri funkcija, njihove vrednosti postaju kopije vrednosti pozivajućih argumenata.

```
public static int obradiZbir (int bF,  
    int aF) {  
    int rF = aF + bF;  
    return rF;  
}  
  
public static void main(String[] args) {  
    int a = 2;  
    int b = 3;  
    int c = obradiZbir(b,a);  
}
```



# Rad sa Stack memorijom – čuvanje promenljivih

```
public static int obradiZbir (int bF,  
    int aF) {  
    int rF = aF + bF;   
    return rF;  
}  
  
public static void main(String[] args) {  
    int a = 2;  
    int b = 3;  
    int c = obradiZbir(b,a);  
}
```

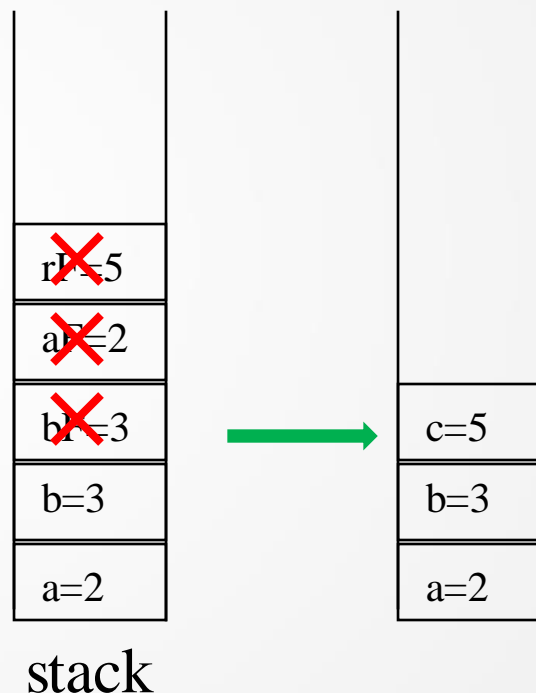


stack

# Rad sa Stack memorijom – čuvanje promenljivih

- ☕ Po završetku funkcije se njene promenljive uklanjaju sa steka

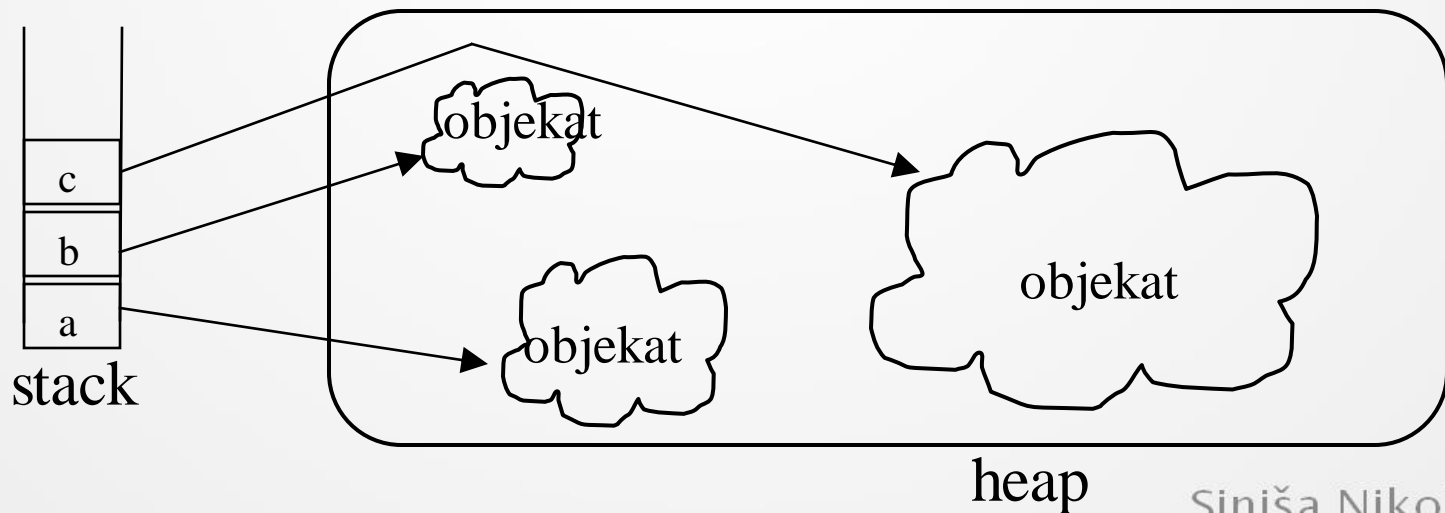
```
public static int obradiZbir (int bF,  
    int aF){  
    int rF = aF + bF;  
    return rF;  
} ←  
public static void main(String[] args) {  
    int a = 2;  
    int b = 3;  
    int c = obradiZbir(b,a) ;  
}
```





# Rad sa Heap memorijom i Objekti u Javi

- ☪ Java svoj rad zasniva na objektima (možete ih zamisliti kao skup promenljivih)
- ☪ Objekti se kreiraju upotrebom ključne reči **new**
- ☪ za čuvanje kreiranih objekata koristi se **heap**
- ☪ na **heap-u** se zauzima (alocira) memorija za objekat, dok se **referenca** (oznaka memorijske lokacije) ka objektu čuva kao vrednost promenljive na **stack-u**



# Rad sa Heap memorijom i Objekti u Javi

- ☞ *Heap* ili dinamička memorija je memorija gde se smeštaju dinamički alocirane vrednosti.
- ☞ Entiteti se modeluju klasama, a instanciranjem tih klasa nastaju objekti
- ☞ Osnovna klasa za sve objekte u Javi je klasa *Object*
- ☞ Sve Java klase direktno ili indirektno nasleđuju klasu *Object* (kasnije objašnjavamo)

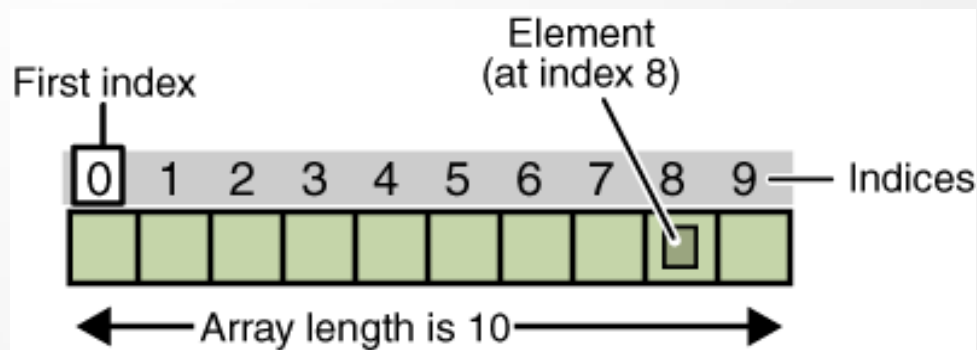
# Rad sa nizovima primitivnih tipova

- ☞ Niz je kontejnerski objekat koji sadrži fiksni broj elemenata istog tipa.
- ☞ kreiraju upotrebom ključne reči **new**
- ☞ Svakom elementu niza pristupa se preko indeksa koji određuje njegovu poziciju u nizu.
- ☞ Indeks prvog elementa niza je 0, a svaki sledeći element poseduje indeks uvećan za jedan.

```
//deklaracija i alokacija  
int imeNiza [] = new int [5];
```

```
//pristupanje članu niza  
imeNiza[indeks]
```

```
//duzina niza  
imeNiza.length
```



# Rad sa nizovima primitivnih tipova

```
int a[]; //isto kao int [] a
```



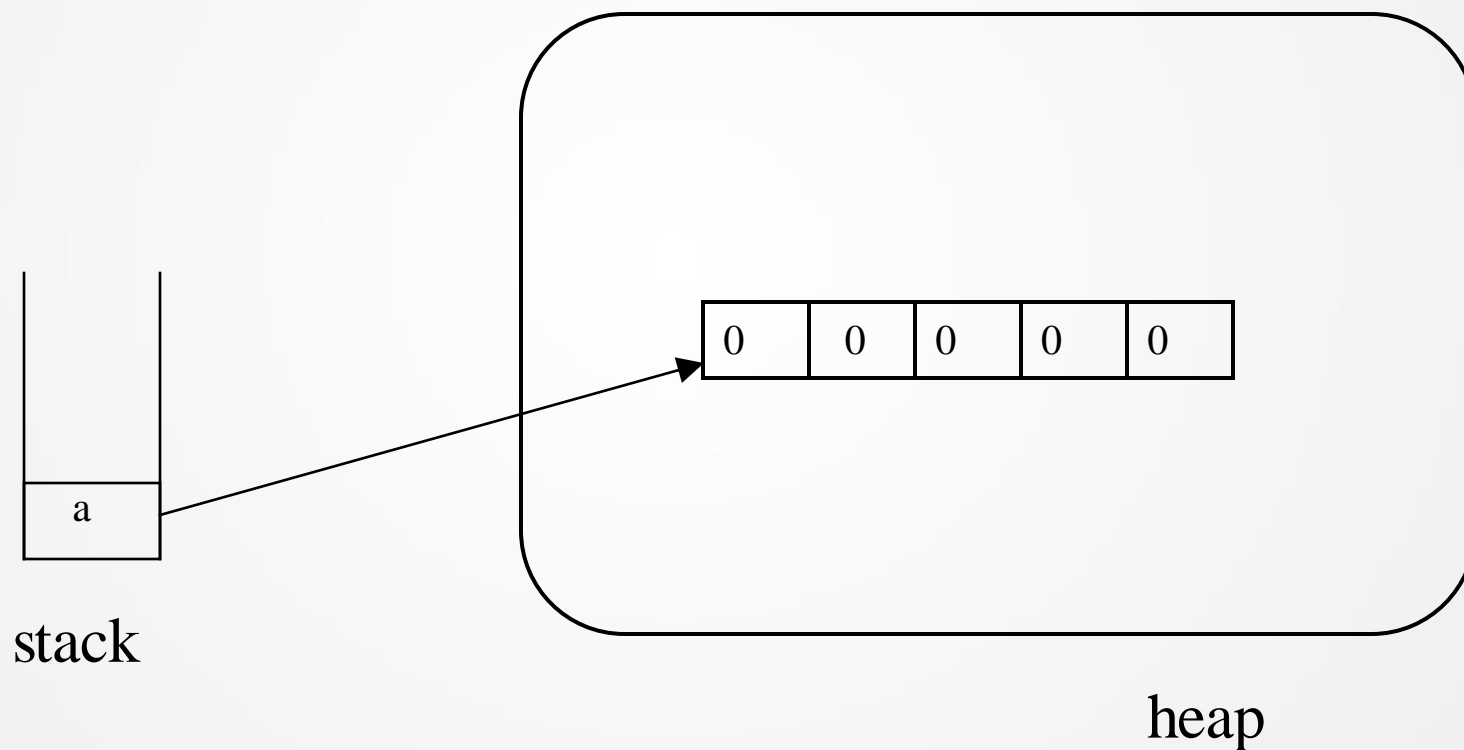
stack



heap

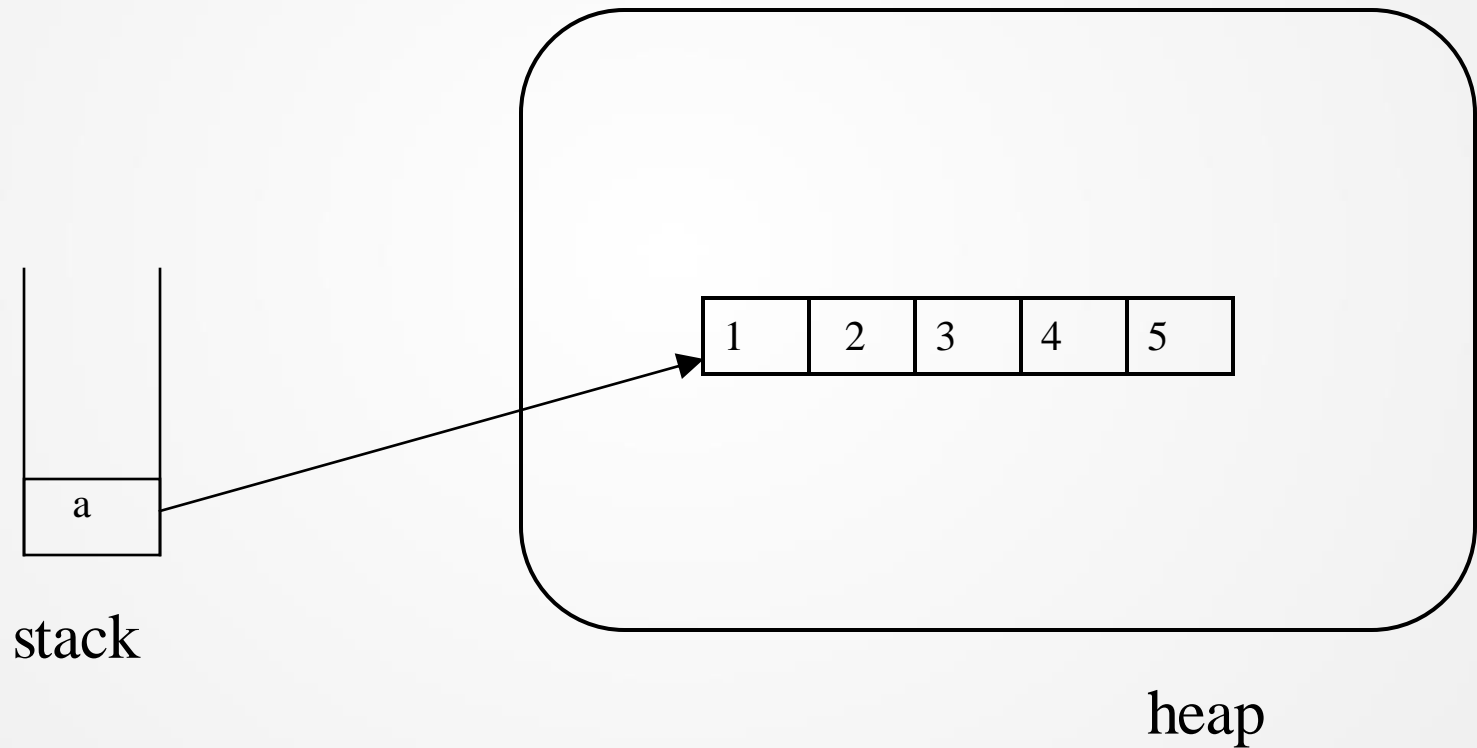
# Rad sa nizovima primitivnih tipova

```
a = new int[5];
```



# Rad sa nizovima primitivnih tipova

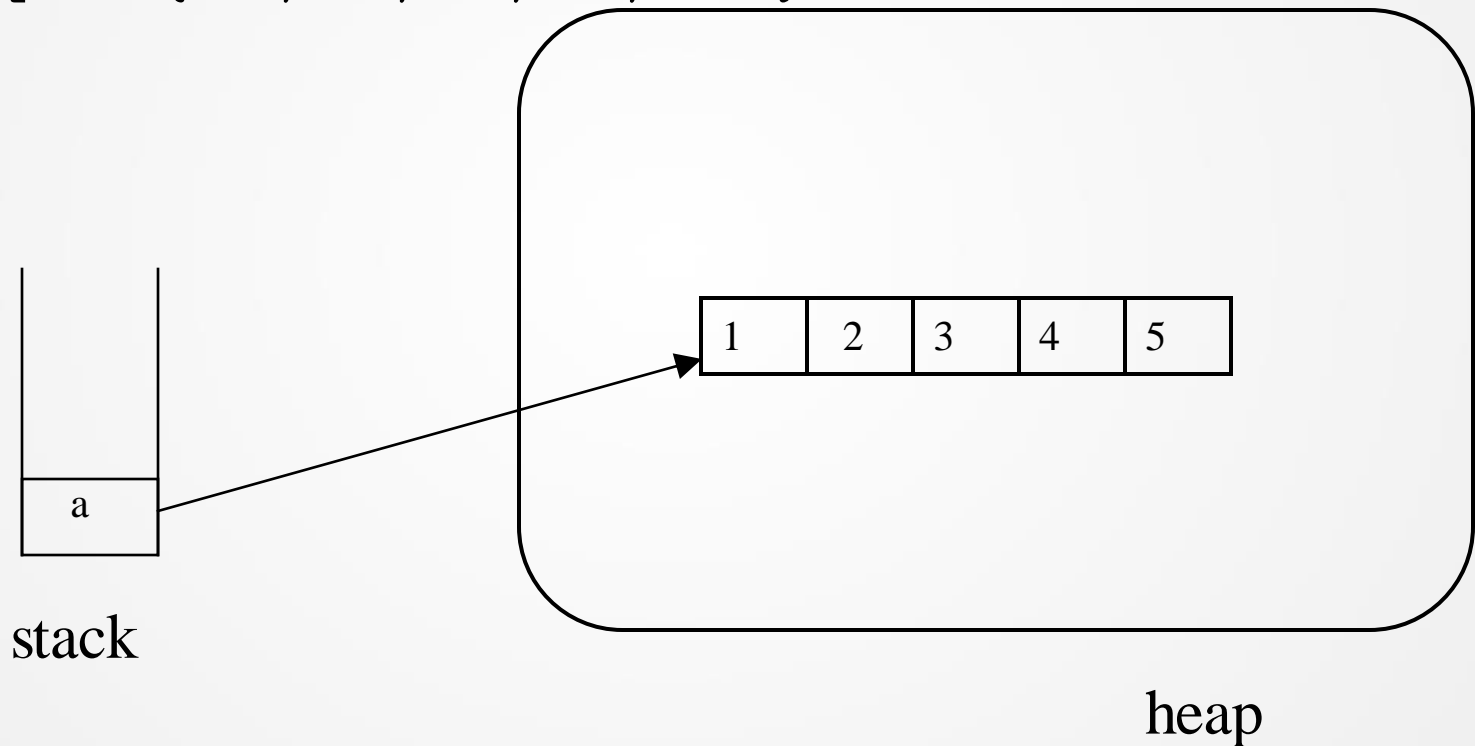
```
a[0]=1; a[1]=2; a[2]=3; a[3]=4; a[4]=5;
```



# Rad sa nizovima primitivnih tipova

- ☕ Niz se može kreirati i popuniti vrednostima u jednoj liniji koda.

```
int a[] = { 1, 2, 3, 4, 5 };
```



# Prolazak kroz elemente niza – for petlja

- ☕ Prolazak redom kroz sve elemente niza može se ostvariti korišćenjem klasične for petlje.
- ☕ Tada, nam je dostupan i indeks elementa niza kojem pristupamo, te možemo menjati vrednost elementa kojem pristupamo.

```
int niz[] = {1, 2, 3, 4};  
for (int i = 0; i < niz.length; i++)  
    System.out.println(niz[i]);
```



# Prolazak kroz elemente niza – for each petlja

- ☞ Za prolazak kroz nizove (i kolekcije, o čemu će biti više reči kasnije) može se koristiti i for petlja za iteriranje (for each petlja).
- ☞ Potencijalni problem: za nizove primitivnih tipova tada se ne može menjati vrednost elementa niza

```
for (promenljiva : niz)  
    telo
```

☞ Primer

```
int niz[] = {1, 2, 3, 4};  
for (int el : niz)  
    System.out.println(el);
```

# Višedimenzionalni nizovi

☞ Višedimenzionalni nizovi se predstavljaju kao nizovi nizova

☞ Može se kreirati i jednim potezom

```
int a[][] = { {1, 2, 3 },  
              {4, 5, 6 } };
```

☞ Mogu se odmah definisati sve dimenzije

```
int a[][] = new int[2][3];
```

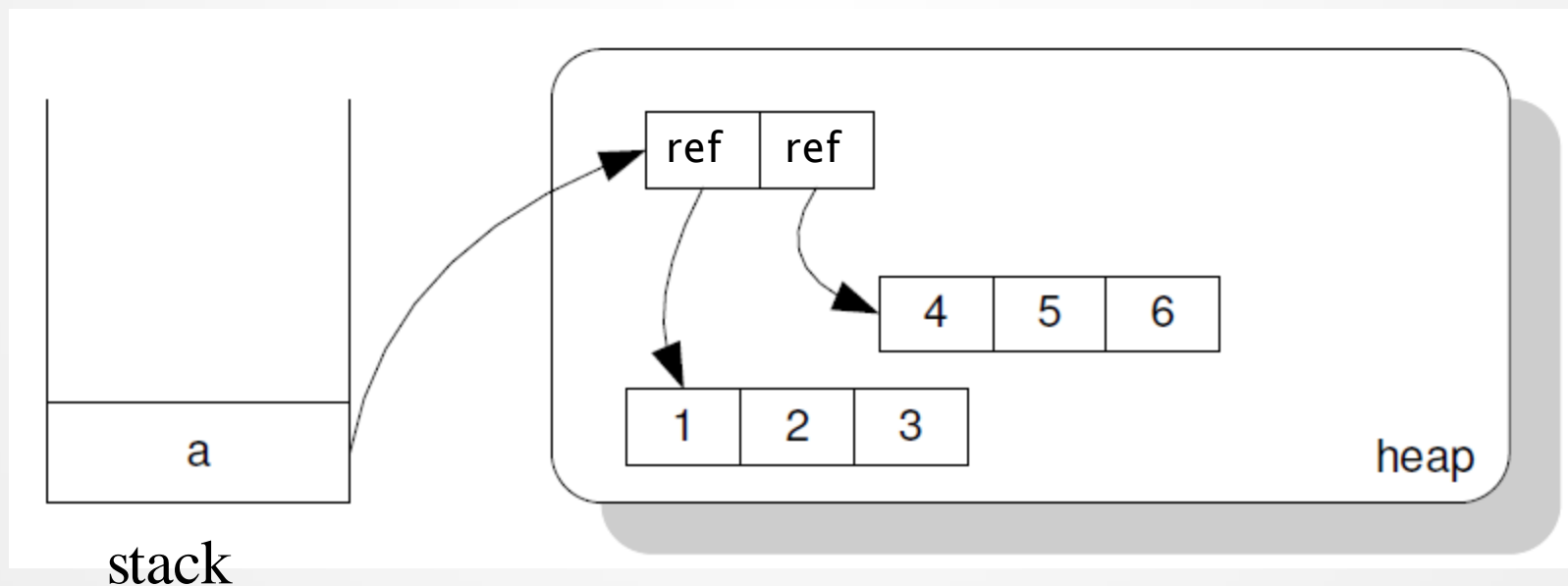
☞ Može se definisati postupno. Odmah se definiše samo prva dimenzija, a druga da se definiše kasnije

```
int a[][] = new int[2][]; //niz ima 3 vrste  
a[0] = new int[3]; //0 vrsta ima 3 kolona  
a[1] = new int[13]; //1 vrsta ima 13 kolona
```

# Višedimenzionalni nizovi

☞ U jednoj liniji kod

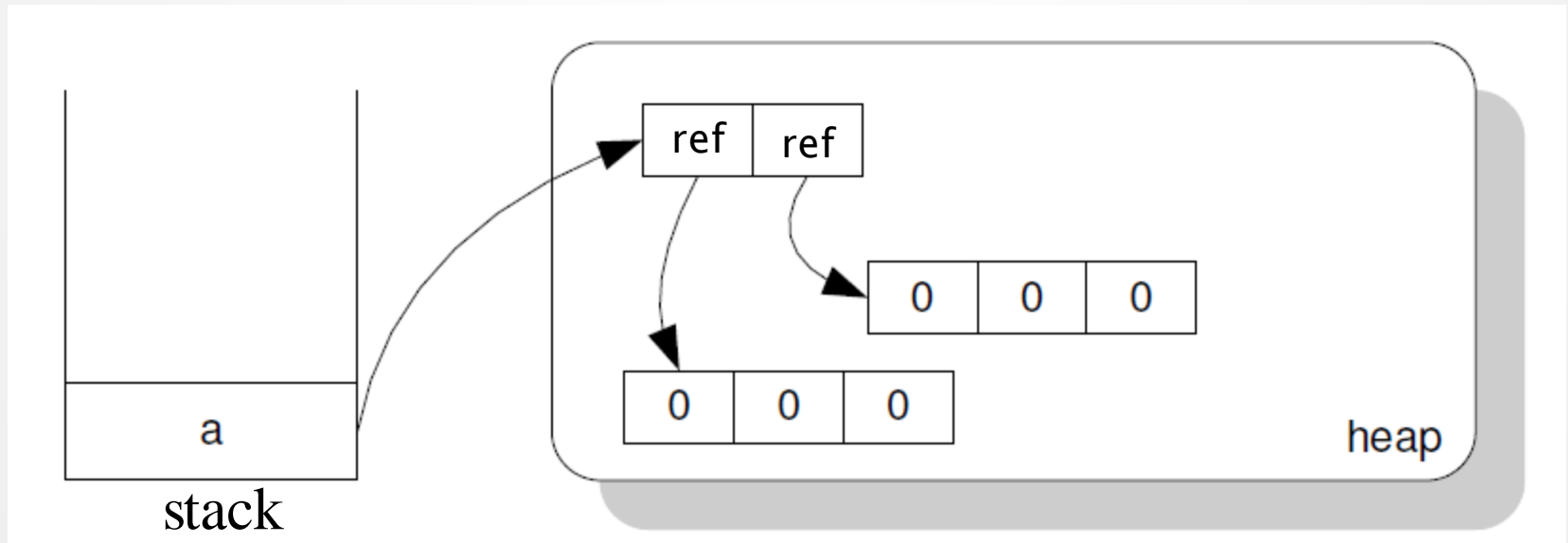
```
int[][] a = { {1, 2, 3}, {4, 5, 6} };
```



# Višedimenzionalni nizovi

☞ Mogu se odmah definisati sve dimenzije:

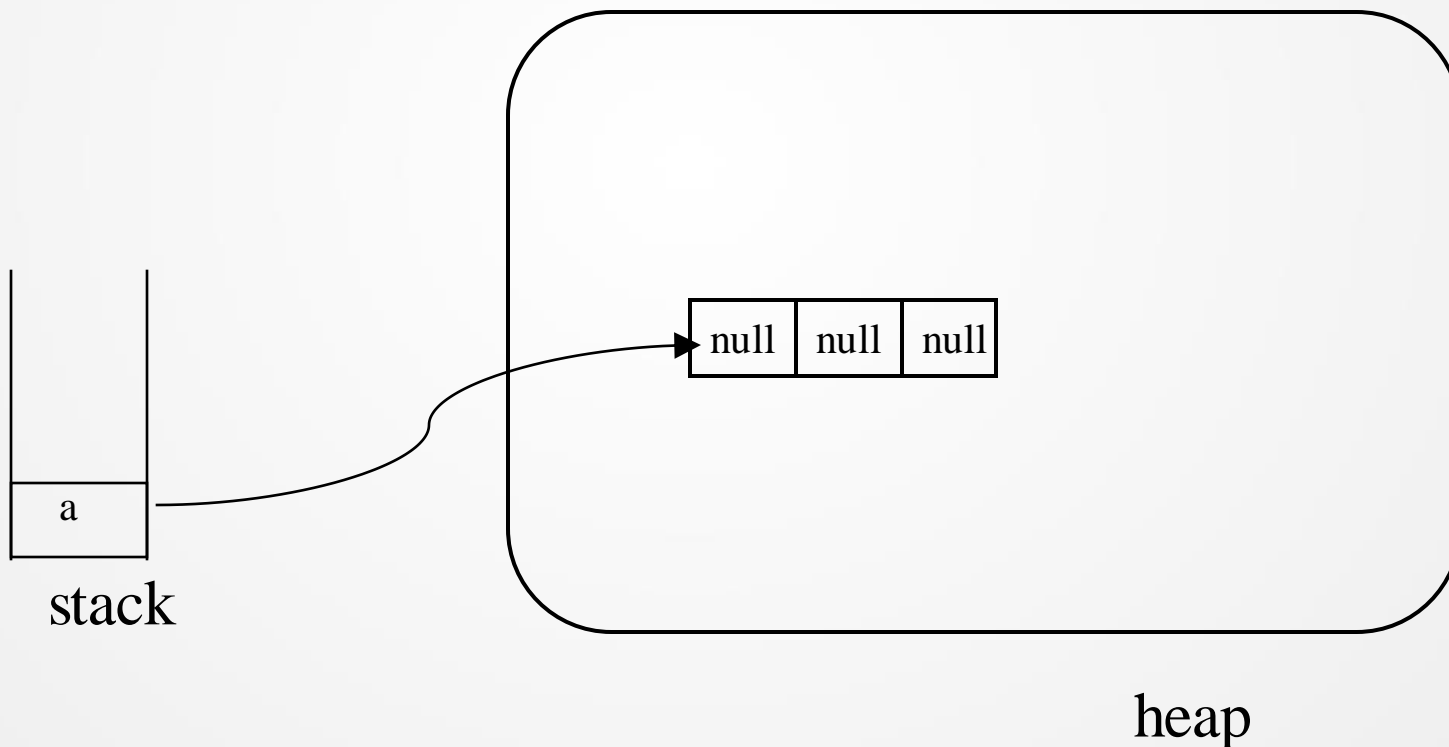
```
int[][] a = new int[2][3];
```



# Višedimenzionalni nizovi

- ☞ Može se definisati postupno. Odmah se definiše samo prva dimenzija, a druga da se definiše kasnije:

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



# Višedimenzionalni nizovi

- ☞ Za svaki element prvog niza definiše se novi niz proizvoljne dužine
- ☞ Moguće je napraviti dvodimenzionalni niz čije vrste imaju različiti broj kolona u svakoj vrsti
- ☞ Jagged array is array of arrays such that member arrays can be of different sizes

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

```
a[0] = new int[1];
```

```
a[0][0]=0;
```

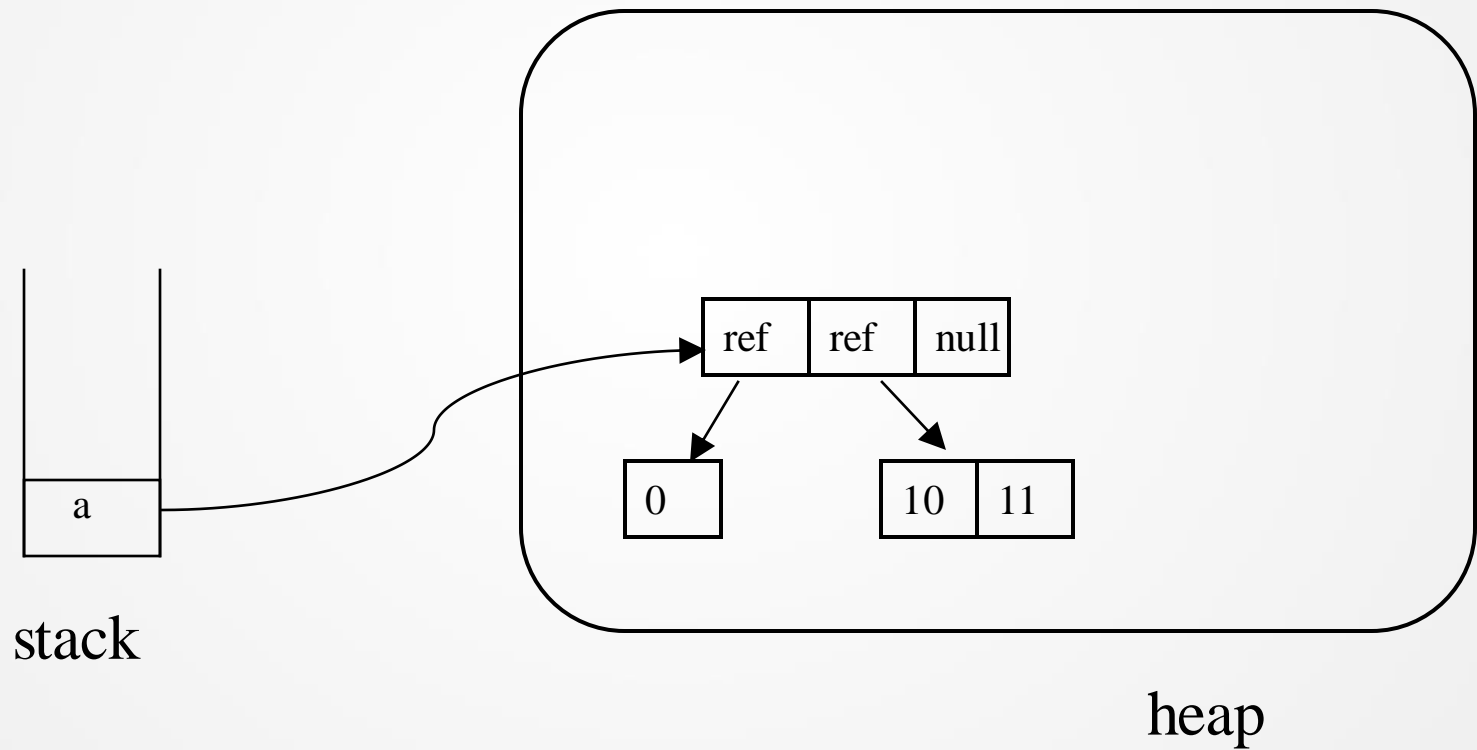
```
a[1] = new int[2];
```

```
a[1][0]=10;
```

```
a[1][1]=11;
```

```
//šta bi bila vrednost a[2] ?
```

# Višedimenzionalni nizovi



# Prolazak kroz elemente višedimenzionalnih nizova

☞ Klasična for petlja

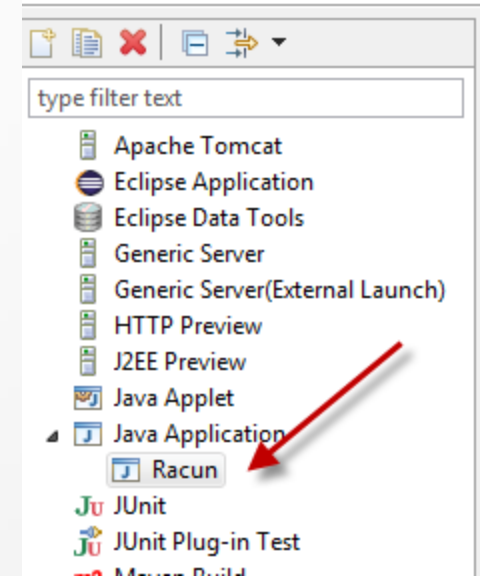
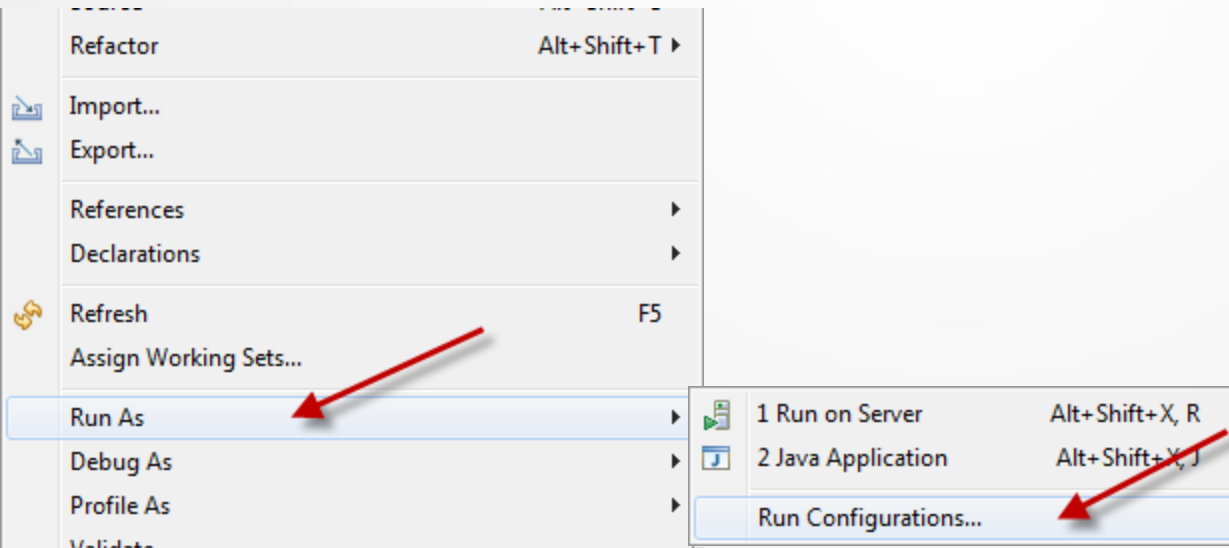
```
int[][] a = { {1, 2, 3}, {4, 5, 6} };  
for (int i = 0; i < a.length; i++) {  
    for (int j = 0; j < a[i].length; j++) {  
        System.out.println(a[i][j]);  
    }  
    System.out.println();  
}
```



# Prosleđivanje parametara prilikom pokretanja java programa iz Eclipse alata

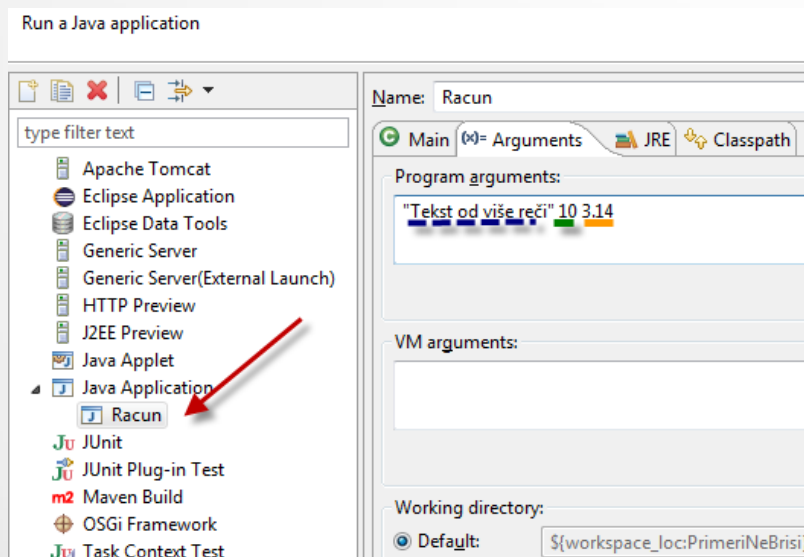
☕ Pevedenom i ranije pokrenutom Java pogramu moguće je proslediti parametre prilikom ponovnog pokretanja

1. Desni klik na klasu -> *Run As* -> *Run Configurations...*  
-> pa iz liste pokrenutih programa selektujete željeni
2. Nad željenim programom sa desne strane odaberite karticu *Arguments* -> *Program Arguments*



# Prosleđivanje parametara prilikom pokretanja java programa iz Eclipse alata

- ☕ Pevedenom i ranije pokrenutom Java pogramu moguće je proslediti parametre prilikom ponovnog pokretanja
  - 3. U delu *Program Arguments* zadaju se parametri poziva aplikacije, pri čemu se svaka reč tretira kao novi argument u pozivu programa
    - a) Tekst se navodi između apostrofa
  - 4. Parametrima se pristupa preko promenljive *args* koja se nalazi u metodi main.

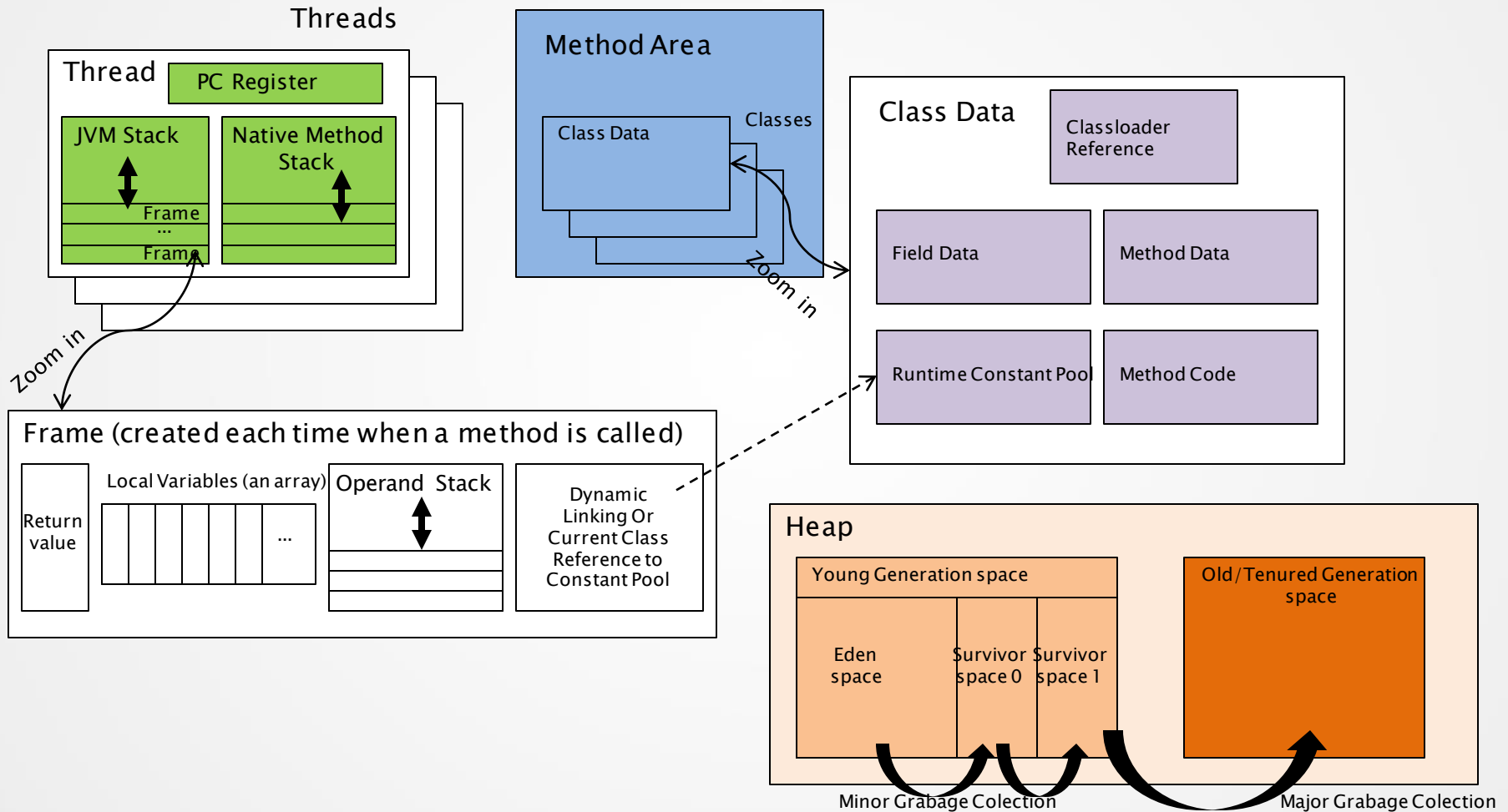


```
public static void main(String[] args) {  
    String tekst = args[0];  
    int broj1 = Integer.parseInt(args[1]);  
}
```

# Java (JVM) Memory Model – Memory Management in Java 1.8

- ☕ Zvanična java dokumentacija navodi nekoliko različitih prostora za skladištenje podataka u toku rada JVM i za izvršavanja Java programa
- ☕ Prostor se deli grubo u dve kategorije
  - ☕ Prostor za podatke koji se kreira za svaku programsku nit (1 Java program može pokrenuti više programskih niti)
  - ☕ Prostor za podatke koji se kreira na nivou JVM (dele ga sve programske niti)

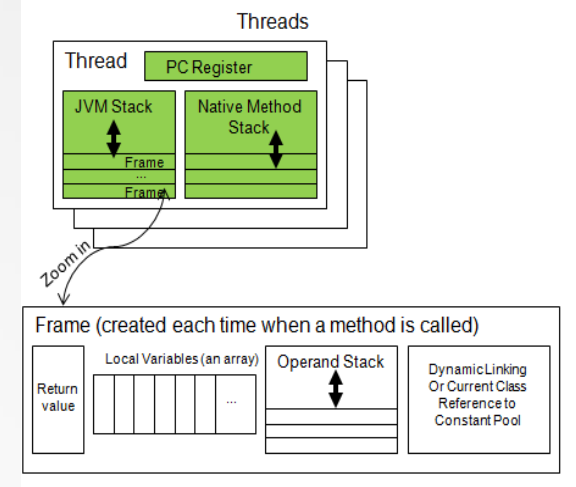
# Java (JVM) Memory Model – Memory Management in Java 1.8



# Memorija za Programske niti

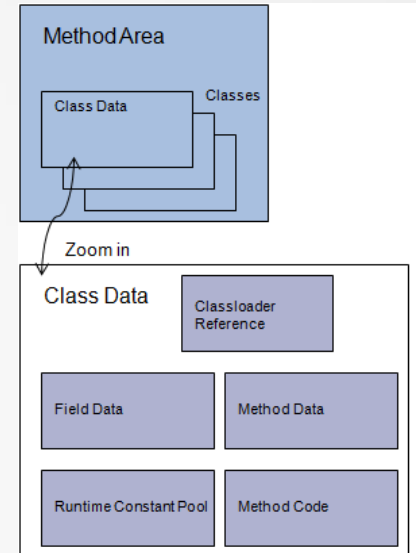
☕ Frejm skladište podatke kao što su:

- povratna vrednost funkcije (*Return value*),
- niz lokalnih promenljivih (*Local Variables*) u redosledu u kojem se pojavljuje u funkciji (pristupa im se na osnovu indeksa, broj promenljivih u nizu i njihove vrednosti determiniran je izvršavanjem metode),
- stek za izvršenje operacija (*Operand Stack*) (prazan pre izvršenja odgovarajuće instrukcije, u toku izvršenja instrukcije popunjava se vrednostima iz lokalnih promenljivih, primenjuje se nad njima odgovarajuća instrukcija, i rezultat instrukcije se stavlja na vrh steka, preuzima se rezultat, ažuriraju se vrednosti lokalnih promenljivih i *Operand Stack* se prazni),
- referenca ka *Runtime Constant Pool* za posmatranu klasu čija se metoda poziva. Referenca omogućava dinamičko povezivanje simboličkih oznaka koje predstavljaju nazive klasa, metoda, atributa, promenljivih..., sa njihovim stvarnim oznakama i vrednostima



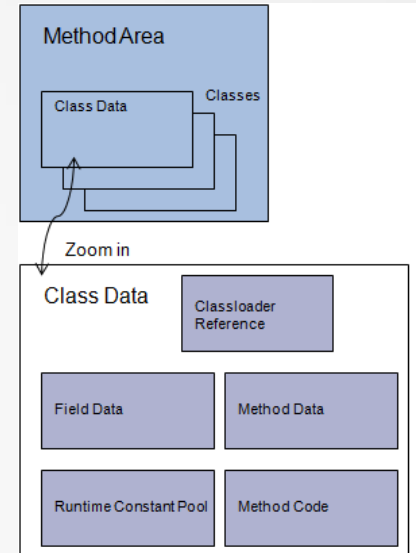
# Deljena Non-Heap memorija – Method Area

- ☞ *Method Area* memorija se deli između svih niti JVM.
- ☞ prostor namenjen za skladištenje metapodataka i informacija za sve klase koje se izvršavaju u aplikaciji (simboličke oznake atributa i metoda, podaci koji definišu strukturu klase i prevedeni programski kod (bytecode) metoda klase,...)
- ☞ Iako se *Method Area* u zvaničnoj *Oracle Java* dokumentaciji definiše kao sastavni deo *Heap* memorije, realna situacija je drugačija i on pripada *non-heap* memoriji. Prethodna tvrdnja se lako može proveriti pokretanjem i praćenjem potrošnje memorije u aplikaciji sa alatom *jconsole* kreiranim za Oracle JVM.
- ☞ *Classloader Reference* sadrži vrednost reference ka nekom od *Classloader* objekata koji je očitao datu klasu (npr. *Bootstrap Classloader*, *Extension Classloader*, *System Classloader*, ...)



# Deljena Non-Heap memorija – Method Area

- ☞ *Field Data* za svaki atribut klase sadrži informacije za ime, tip, modifikatore pristupa i dodatne vrednosti.
- ☞ *Method Data* za svaku metodu sadrži informacije za ime, povratnu vrednost, tipove parametara, modifikatore pristupa i dodatne attribute
- ☞ *Method Code* za svaku metodu sadrži prevedeni kod (*bytecode*), količinu memorije potrebnu za *Operand Stack*, količinu memorije potrebnu za *Local Variables*, tabela lokalnih promenljivih, tabela numeričkih oznaka za liniju u java kodu koja odgovara instrukciji iz prevedenog koda (za debugovanje), tabela izuzetaka koji se mogu javiti u kodu.

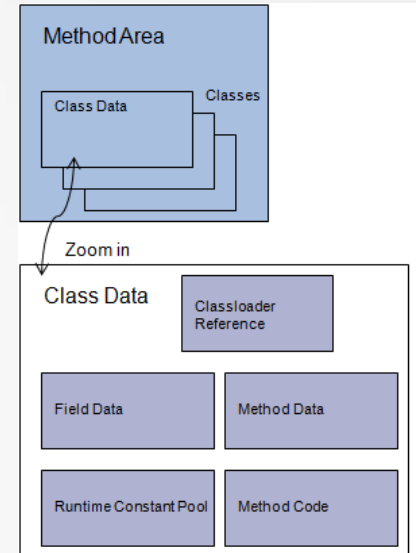


# Deljena Non-Heap memorija – Method Area

☞ *Runtime Constant Pool* sadrži simboličke oznake i vrednosti vezane za te oznake. U prevedenom programskom kodu umesto naziva klasa, metoda, atributa, string vrednosti zadatih direktno u kodu, integer vrednosti zadatih direktno u kodu, double vrednosti zadatih direktno u kodu,..., koristi se za njih definisana simbolička oznaka, a prava vrednost se preuzima iz *Runtime Constant Pool*. Prethodno se zove dinamičko povezivanje i neophodno je jer se u prevedenom kodu ne skladište veliki podaci (referenca ka određenoj metodi, String tekstualna vrednost “Hello” koja je direktno zadata u kodu).

☞ Posmatrajući java kod klase MojaKlasa

```
package SinisinPaket;  
  
public class MojaKlasa {  
  
    public static void main(String[] args) {  
  
        System.out.println("String vrednost zadata direktno u kodu");  
    }  
}
```





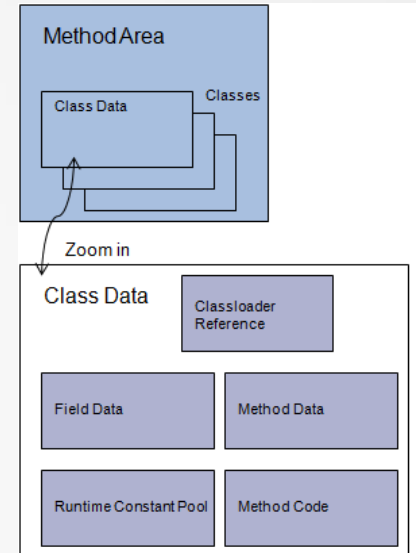
# Deljena Non-Heap memorija – Method Area

Možemo da izvršimo inspekciju strukture prevedenog koda `MojaKlasa.class` naredbom

`javap -v -p -s -sysinfo -constants classes/SinisinPaket/MojaKlasa.class`

Classfile /c:/DoobukaJWTS/PrimeriNeBrisi/bin/SinisinPaket/MojaKlasa.class

```
Last modified Dec 2, 2016; size 583 bytes
MD5 checksum aa5b26ab5a04e7e66138c06710368abd
Compiled from "MojaKlasa.java"
public class SinisinPaket.MojaKlasa
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
  Constant pool:
    #1 = Class                #2          // SinisinPaket/MojaKlasa
    #2 = Utf8                  SinisinPaket/MojaKlasa
    #3 = Class                #4          // java/lang/Object
    #4 = Utf8                  java/lang/Object
    #5 = Utf8                  <init>
    #6 = Utf8                  ()V
    #7 = Utf8                  Code
    #8 = Methodref             #3.#9      // java/lang/Object.<init>:()V
    #9 = NameAndType           #5:#6      // "<init>":()V
    #10 = Utf8                 LineNumberTable
    #11 = Utf8                 LocalVariableTable
    #12 = Utf8                 this
    #13 = Utf8                 LSinisinPaket/MojaKlasa;
    #14 = Utf8                 main
    #15 = Utf8                 ([Ljava/lang/String;)V
    #16 = Fieldref             #17.#19    // java/lang/System.out:Ljava/io/PrintStream;
    #17 = Class                #18        // java/lang/System
    #18 = Utf8                 java/lang/System
    #19 = NameAndType           #20:#21    // out:Ljava/io/PrintStream;
    #20 = Utf8                 out
    #21 = Utf8                 Ljava/io/PrintStream;
    #22 = String               #23        // String vrednost zadata direktno u kodu
    #23 = Utf8                 String vrednost zadata direktno u kodu
    #24 = Methodref             #25.#27    // java/io/PrintStream.println:(Ljava/lang/String;)V
```

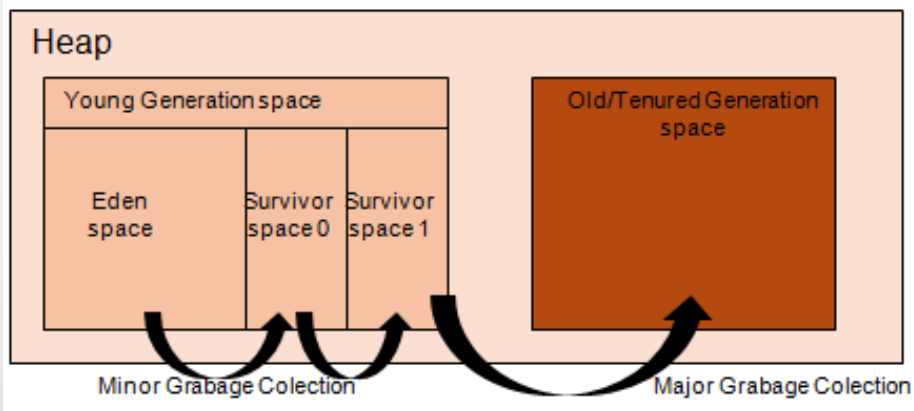


Vidimo da *Constant pool* sadrži simboličke oznake koje predstavljaju sa znakom # i brojem, iza kojih se navodi njihova konkretna vrednost

Simbolička oznaka #23 sadrži vrednost testa “String vrednost zadata direktno u kodu”

# Deljena Heap memorija

- ☞ *Heap* (dinamička memorija) deo memorije se deli između svih niti JVM
- ☞ je memorija gde se smeštaju dinamički alocirane vrednosti (vrednosti mogu da menjaju veličinu)
- ☞ smeštaju se vrednosti instanca klasa i nizovi
- ☞ Vrednosti sa *heap*-a se ne brišu kada se završi metoda (kao na steku), već njih uklanja poseban proces zvan ***garbage collector***(gc)
- ☞ Svi novi objekti i nizovi se kreiraju u *Eden space*-u dela *Young Generation space*-u
- ☞ Kada se *Eden space* popuni, pokreće se brisanje neiskorišćenih objekata, prostor *Eden space* postaje prazan, a svi objekti koji su preživeli brisanje premeštaju se u *Survivor space 0*.
- ☞ Na sličan način funkcioniše brisanje objekata u *Survivor Space 0* i *1*.

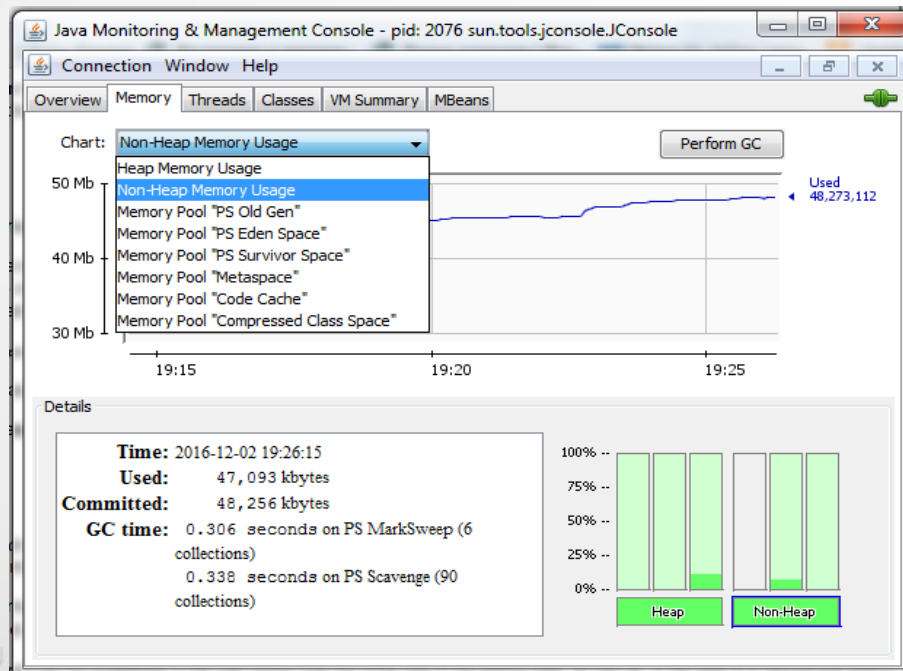


- ☞ Kada se *Survivor Space 0* popuni, pokreće se brisanje neiskorišćenih objekata, prostor *Survivor Space 0* postaje prazan, a svi objekti koji su preživeli brisanje premeštaju se u *Survivor space 1*.

- ☞ Brisanjem objekta u *Survivor space 1*, preživeli objekti završavaju u *Old/Tenured Generation space*-u.

# Pogled na memoriju iz alata jconsole

- ☞ Kao već navedeno deljena JVM memorija se deli u dve grupe *Heap* i *Non-Heap* memoriju.
- ☞ Izgled *Heap* memorije podudara se sa onim iz dokumentacije, dok se izgled *Non-Heap* memorije malo razlikuje.
- ☞ *Non-Heap* memorija obuhvata deo *Metaspace* (u javi 1.7 je to bio *Permanent Generation*), *Code Cache* i *Compressed Class Space*.
- ☞ *Metaspace* deo memorije sadrži *Method Area* deo tj. koristi se za skladištenje metapodataka o klasama. To se nalaze i različiti *Memory Pool*-ovi.



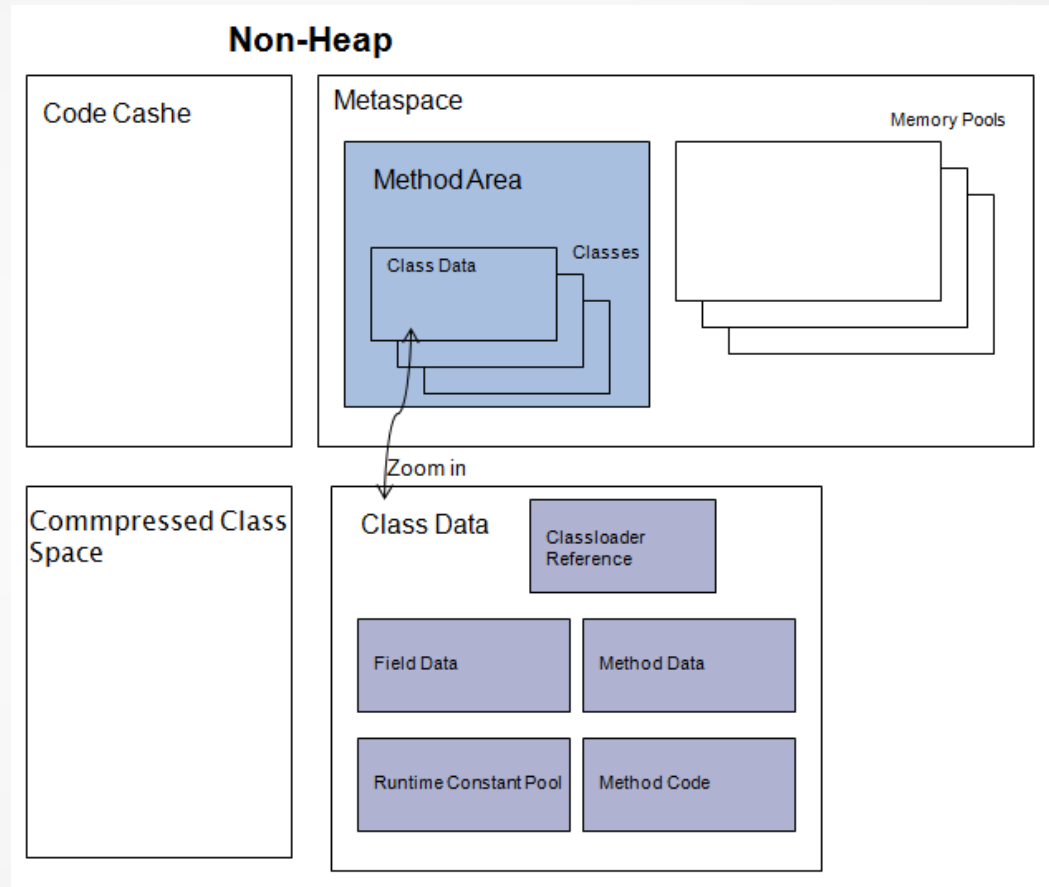
☞ *Compressed Class Space* se takođe koristi za skladištenje metapodataka o klasama.

☞ *Code Cache* memorija se koristi za kompajliranje i skladištenje koda koji je kompajliran u native programski kod.

# Pogled na memoriju iz alata jconsole



Kompletniji prikaz Non-Heap Menorije uključivši informacije iz alata jconsole bi bio:



# Oslobađanje neiskorišćene memorije u Javi

- ☕ *Garbage collector* radi kao poseban proces u pozadini
- ☕ Automatska dealokacija memorije
- ☕ Automatska defragmentacija memorije
- ☕ Sistem ga poziva se po potrebi
- ☕ Korisnik ga može eksplicitno pozvati kodom:  
**`System.gc()`** ; ali će *Garbage Collector* sam "odlučiti" da li će pokrenuti proces oslobađanja memorije. Poziv ove metode je samo sugestija GC-u da bi mogao da otpočne čišćenje. I pored *Garbage Collector*-a može doći do greške *OutOfMemory* ako ne vodimo računa

# Razlike između Heap i Stack memorije

1. *Heap* se deli između svih niti JVM, dok se *stack* koristi za tačno određenu nit.
2. *Stack* skladišti vrednosti za lokalne promenljive primitivnih tipova i reference za lokalne promenljive koje pokazuju na objekte u *heap*-u. *Heap* ne skladišti vrednosti lokalnih promenljivih, već se na njemu skladište java objekti.
3. Objekti koji se skladište na *heap*-u su globalno dostupni preko njihove reference, dok su *stack* vrednosti dostupne samo za određenu nit.
4. *Stack* nije izdelfen na delove i upravljanje memorijom na *stack*-u je po *LIFO (Last-In-First-Out)* principu, dok je *heap* memorija izdelfjena na delove, te je upravljanje memorijom *heap*-a kompleksnije i ostavljeno posebnom procesu nazvanom *Garbage Collector*.
5. *Stack* memorija je kratkog veka (traje koliko i nit), dok *heap* memorija traje od paljenja do gašenja JVM.
6. *Stack* memorija je znatno manja u veličini u poređenju sa *heap* memorijom.
7. Kada se *stack* popuni greška je *java.lang.StackOverflowError* dok kada se *heap* popuni greška je *java.lang.OutOfMemoryError: Java heap space*.