



**ftn**informatika

# *Java Web Development*

Modul 1

Termin 2

# Sadržaj

1. Promenljive (podsetnik)
2. Nizovi
3. Klasa String
4. Osnovni koncepti u OOP
5. Osnovni principi u OOP
6. Klase/Objekti (konstruktori, modifikatori pristupa, get/set, null/this, metode)
7. Čuvanje promenljivih - rad sa stack memorijom
8. Čuvanje promenljivih - rad sa heap memorijom
  
9. Dodatni materijali

# Promenljive

- U Javi postoje dva tipa podataka:

Osnovni tip promenljivih	Složeni tip promenljivih
byte, short, int, long, float, double, boolean	objekti

- Složene tipove podataka možete zamisliti kao skup promenljivih, sastavljen od primitivnih ili složenih promenljivih.

# Nizovi

- 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];`

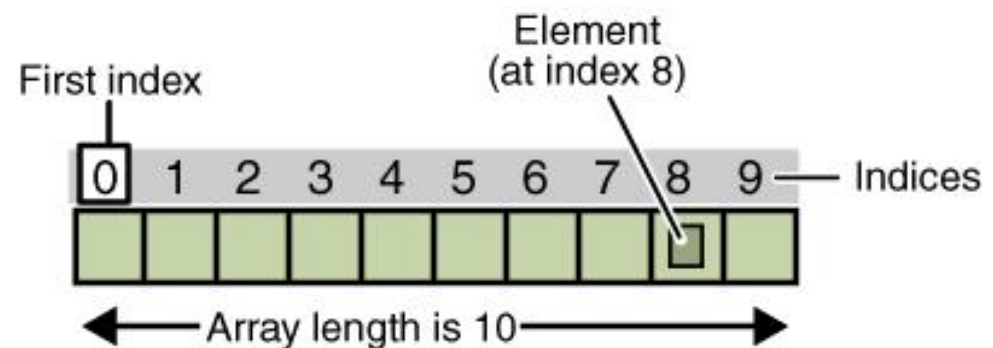
- `//pristupanje članu niza`

- `imeNiza[indeks]`

- 

- `//duzina niza`

- `imeNiza.length`



# Nizovi – načini instanciranja i dodele

Prvi način:

- `int a[] = new int[3];`  
`a[0] = 1; a[1] = 2; a[2] = 3;`

Drugi način:

- `int b[] = {1, 2, 3};`

# Nizovi – prolazak kroz elemente niza

Prvi način (u nastavku kursa **ne** raditi sa ovom petljom):

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

```
for (int i = 0; i < niz.length; i++)  
    System.out.println(niz[i]);
```



Klasična for petlja.

Drugi način (Od sada smo fokusirani na ovaj način):

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

```
for (int el : niz)  
    System.out.println(el);
```



for each petlja.

# Klasa String

- Niz karaktera je podržan klasom String. String nije samo niz karaktera – **on je klasa!**
- Od java 1.7 skladište se na heap memoriji u delu *String pool*.
- **Objekti klase String se ne mogu menjati (*immutable*)!**
- *Immutable Objects* je objekat kome se definiše vrednost u trenutku njegovog kreiranja. Za njega ne postoje metode, ni načini kako da se ta vrednost dodatno promeni.

# Klasa String

- Za cast-ovanje String-a u neki primitivni tip koristi se Wrapper klasa (učimo na predstojećim terminima) i njena metoda `parseXxx()`:

**int** i = Integer.parseInt(s);

- Za poređenje Stringova se ne koristi operator `==`, već funkcija **equals** ili **equalsIgnoreCase**

- Reprezentativne metode

- `str.length()`
- `str.charAt(i)`
- `str.indexOf(s)`
- `str.substring(a,b)`, `str.substring(a)`
- `str.equals(s)`, `str.equalsIgnoreCase(s)` – ne koristiti `==`
- `str.toLowerCase()`



# Klasa String

```
String s1 = "Ovo je";  
String s2 = "je string";  
System.out.println(s1.substring(2)); // o je  
System.out.println(s2.charAt(3)); // s  
System.out.println(s1.equals(s2)); //false  
System.out.println(s1.indexOf("je")); // 4 , ako nema podstringa vratiće  
-1  
System.out.println(s2.length()); //9  
System.out.println(s2.startsWith("je")); //true
```

# Klasa String – metoda split()

- Metoda *split* "cepa" osnovni string na niz stringova po zadatom šablonu
  - originalni string se ne menja
  - parametar je regularni izraz
- rezultat je niz stringova na koje je „pocepan“ originalni string
- Poziv:  
**String[] rez = s.split("regex");**

# Klasa String – metoda split()

```
class SplitTest {  
    public static void main(String args[]) {  
        String text = "Ovo je probni tekst";  
        String[] tokens = text.split(" ");  
        for (String i : tokens)  
            System.out.println(i);  
    }  
}
```

Konzola

Ovo

je

probni

tekst

# Zadatak - Zajedno radimo

## 1. Zadatak:

Napraviti klasu Test sa main funkcijom. Splitovati date stringove po odgovarajućem delimiteru (regeksu) i ispisati date reči:

"Pera;Peric;11.06.2011./Zika;Zikic;11.06.2011."

Kako biste grupisali oznake: Pera, Perić, 11.06.2010, Žika, Žikić, 12.07.2011. tako da se nalaze na jednom mestu u memoriji? Kako ćemo znati odakle obeležja za jedan entitet počinju a odakle se završavaju?

- Nizovima?

```
String[][] osoba = new String[2][3];  
osoba[0][0] = "Pera";  
osoba[0][1] = "Perić";  
osoba[0][2] = "11.06.1990";
```

- Mnogo prirodnije bi bilo da sve podatke o jednom entitetu grupišemo i postavimo na jednom mestu. Rešenje bi bilo koristiti Klase.

# Osnovni principi OOP u Javi

1. Enkapsulacija
2. Apstrakcija
3. Nasleđivanje
4. Polimorfizam

# Osnovni koncepti OOP u Javi

- Temin *object-oriented programming* (OOP) predstavlja način razmišljanja za rešavanje programerskih problema.
- Srž ovog načina razmišljanja čini koncept objekta.
- Neki ovaj način razmišljanja nazivaju još i *class-oriented programming*, zato što se za opis objekta koriste klase.
- U OOP-u, u sklopu opisa problema, potrebno je uočiti entitete (jedinice posmatranja) koji se nalaze u realnom svetu (domenu) u kojem se nalazi problem koji se rešava.
- Entiteti se modeluju klasama, a instanciranjem tih klasa nastaju objekti. Odnosno mi objekte opisujemo klasama.



# Osnovni koncepti OOP u Javi

- Da bi se podržalo rešavanje određenog problema, potrebno je uočene entitete opisati na nekakav način i navesti operacije nad tim entitetima i njihovim osobinama (metode).
- Objektno orijentisano programiranje se svodi na identifikaciju entiteta u nekom domenu, navođenje njihovih osobina i pisanje operacija nad tim osobinama.
- U objektno orijentisanoj terminologiji, entiteti su opisani klasama, osobine su atributi, a operacije su metode.
- Osnovni koncepti u javi su **KLASA** i **OBJEKAT**.

# Osnovni koncepti OOP u Javi

- Primer preslikavanja realnog sveta na programiranje:

- U realnom svetu:

- Entitet: Mačka
    - Osobine: Boja očiju, boja dlake, ime
    - Konkretna mačka: Plava, bela, Flekica



- U programiranju:

- Klasa: Mačka
    - Atributi: boja očiju, boja dlake, ime
    - Konkretna instanca klase (Objekat): Plava, bela, Flekica



```
class Macka{  
    String bojaOciju;  
    String bojaDlake;  
    String ime;  
}
```

```
Macka m = new Macka("plava",  
    "bela", "Flekica");
```

# Klase

- Klasu posmatramo kao šablon kojim kreiramo objekte.
- Klase se dizajniraju tako da sadrže sve najbitnije osobine entiteta.
- Klasa predstavlja model objekta i uključuje attribute i metode .
- U Javi je sve predstavljeno klasama, sve što napišemo (metode, promenljive...) mora se naći u okviru neke klase tj. nije moguće definisati funkcije i promenljive izvan neke klase.
- I *main* metoda se nalazi u nekoj klasi!!!
- Listing za definisanje klase počinje ključnom reči **class** .

# Klase

- Primer klase
  - Za Bioskop aplikaciju potrebno je modelovati entitet Karta.
    - Karta je opisana id-jem, datumom i vremenom, salom, tipom i cenom karte.
    - Moguća ponašanja entiteta je prodaja karata.
- Na slici je prikazan izgled klase koji je modelovan alatom draw.io.



# Klase

Naziv klase koja će  
sadržati main funkciju

Naziv klase koja će  
opisivati entitet Karta i  
sadržati odgovarajuće  
atribute

- Primer klase

- Za Bioskop aplikaciju potrebno je modelovati entitet Karta.
  - Karta je opisana id-jem, datumom i vremenom, salom, tipom i cenom karte.
  - Moguća ponašanja entiteta je prodaja karata.

- Na slici je prikazan izgled klase koji je modelovan alatom draw.io.



# Klase

- Primer klase u kodu:

```
class Karta{  
    long id;  
    String datumIVreme;  
    int sala;  
    String tip;  
    double cenaKarte;  
    void prodajaKarata(){  
        System.out.println("Počela je prodaja karata");  
    }  
}
```

Karta
- id: long
- datumIVreme: LocalDateTime
- sala: int
- tip: String
- cenaKarte: double
+ prodajaKarata()

# Objekti - instanciranje klase

- Java svoj rad zasniva na objektima
- Objekti se kreiraju upotrebom ključne reči **new**.
- Nakon ključne reči new pišemo odgovarajući konstruktor (nešto više o njima na kraju termina).

Karta karta = **new** Karta();

Defaultni-  
osnovni  
konstruktor.

# Atribut u klasi **primitivnog** tipa

- Primer kreiranje (instanciranje) objekta

```
class Test {  
    public static void main(String args[]) {  
        Karta karta;  
        karta = new Karta();  
        karta.id = 0;  
        karta.datumIVreme = "10.11.2021.";  
        karta.sala = 1;  
        karta.tip = "2D";  
        karta.cenaKarte = 1200.1;  
        karta.prodajaKarata();  
    }  
}
```

//instanciranje klase

//postavljanje vrednosti atributa

//pozivanje metode



# Atribut u klasi **primitivnog** tipa

- Primer kreiranje (instanciranje) objekta

```
class Test {  
    public static void main(String args[]) {  
        Karta karta;  
        karta= new Karta();  
        karta.id = 0;  
        karta.datumIVreme = "10.11.2021.";  
        karta.sala = 1;  
        karta.tip = "2D";  
        karta.cenaKarte =1200.1;  
        karta.prodajaKarata();  
    }  
}
```

Postavljamo  
vrednosti u  
datom objektu.

Instanciramo  
konstruktorom.

//instanciranje klase

//postavljanje vrednosti atributa

//pozivanje metode

Metoda se  
poziva nad  
objektom.

# Atribut u klasi **složenog** tipa

- Klasa takođe može imati atribut koji nije primitivni tip tj. atribut može sadržati složeni tip(referencu ka nekom objektu) ili više složenih tipova(kolekciju referenci ka nekom objektu).

```
public class Karta{  
    Film film;  
}
```



Referenca ka objektu klase  
Film

# Atribut u klasi **složenog** tipa

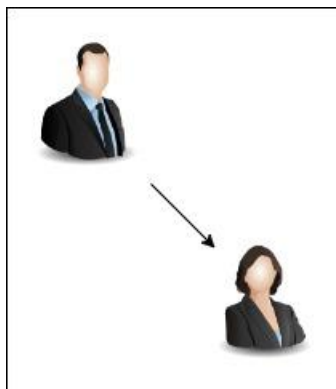
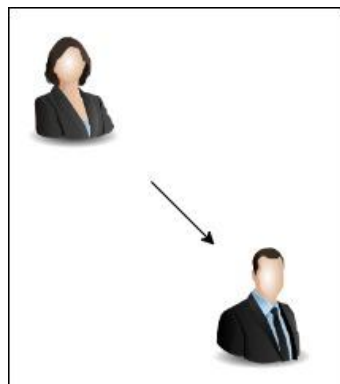
- Atributi koji nisu primitivni tipovi imaju podrazumevanu vrednost **null** što može izazvati **NullPointerException** grešku u radu aplikacije (runtime exception).

# Atribut u klasi **složenog** tipa

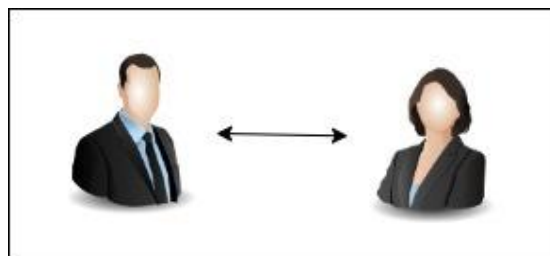
- Veze mogu biti:
  - jednosmerne
  - dvosmerne

# Atribut u klasi **složenog** tipa

- Jednosmerne (unidirekcione)



- Dvosmerne (bidirekcione)



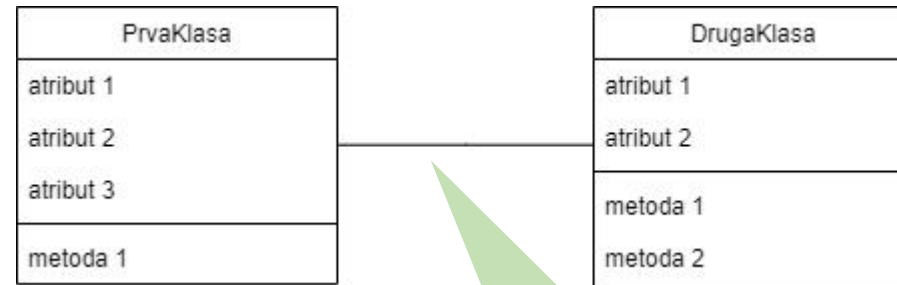
# Atribut u klasi **složenog** tipa

- Klase mogu biti opisane tekstom ili klasnim dijagramom.
- Atributi koji nisu primitivni tipovi mogu biti referenca na drugu klasu ili skup (kolekcija) referenci ka drugoj klasi.
- Klasni dijagram (nacrtan u draw.io editoru):



# Atribut u klasi **složenog** tipa

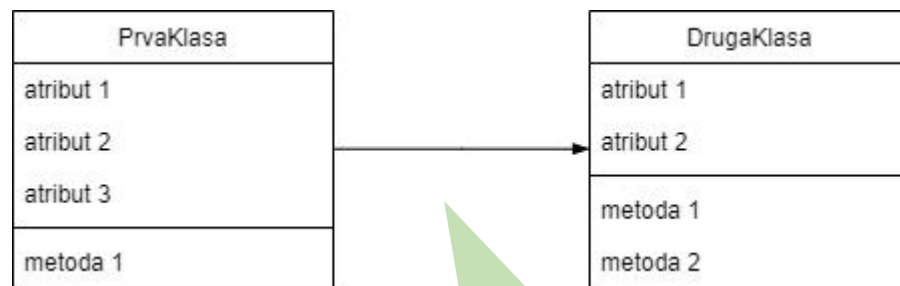
- Veza između klasa
  - dvostruka veza



Dvostruka veza označava da iz prve klase možemo pristupiti drugoj i suprotno.

# Atribut u klasi **složenog** tipa

- Veza između klasa
  - jednostruka veza

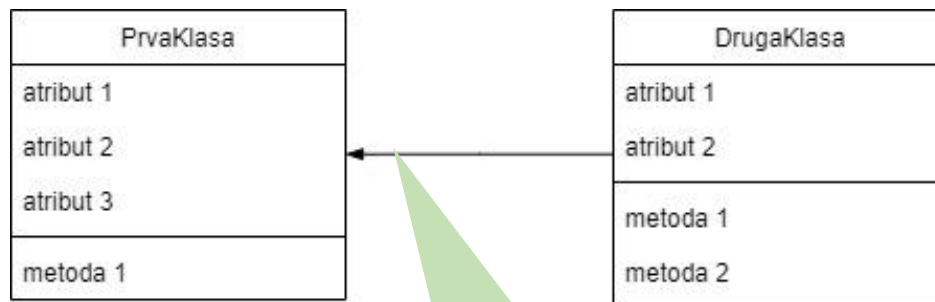


Jednostruka veza označava da iz PrveKlase možemo pristupiti drugoj klasi, ali suprotno ne važi.



# Atribut u klasi **složenog** tipa

- Veza između klasa
  - jednostruka veza



Jednostruka veza označava da iz DrugeKlase možemo pristupiti prvoj klasi, ali suprotno ne važi.

# Atribut u klasi **složenog** tipa

- Veza između klasa
  - Razjasnili smo vezu asocijacije. Pristup u klasama. Neophodno je znati koliko se objekata te klase povezuje sa 1 objektom druge klase. Opisi mogu biti (nazivaju se i **kardinaliteti**):
    - (1..1) tačno jedan,
    - (0..1) jedan ili nijedan
    - (0..\*) više
    - (1..\*) bar jedan

# Atribut u klasi **složenog** tipa

- Veza između klasa
  - kardinaliteti

Da li mi moramo postaviti vrednost složenog atributa ili ne?

0 -> ne moramo  
1 -> moramo

1.....	1
0.....	1
0.....	*
1.....	*

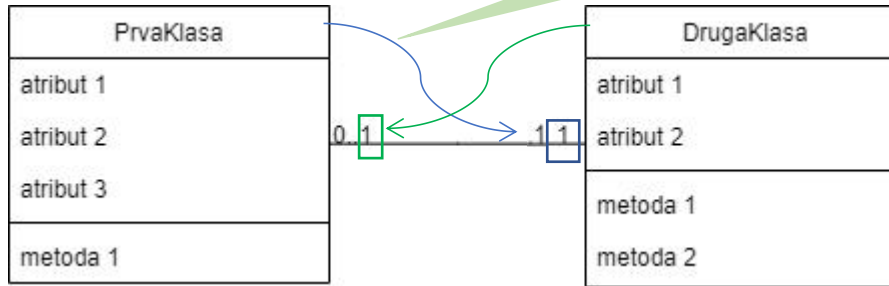
Označava da li je neophodno postojanje objekta sa druge strane veze. Ovaj broj se gleda kada instanciramo objekte u Testnoj klasi.

Označava da li ćemo u klasi kao atribut staviti referencu (1) ili skup referenci (\*). Ovaj broj se gleda kada pišemo attribute Entitet klase.

# Atribut u klasi **složenog** tipa

Kardinalitet za datu klasu  
čitamo sa suprotne strane

- Veza između klasa
  - dvostruka veza



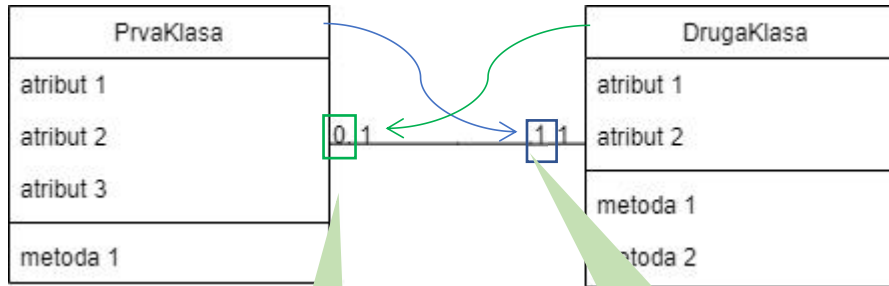
```
class PrvaKlasa{  
    DrugaKlasa referenca1;  
}
```

```
class DrugaKlasa{  
    PrvaKlasa referenca2;  
}
```

# Atribut u klasi **složenog** tipa

- Veza između klasa
  - dvostruka veza

```
class MainKlasa{  
    public static void main(String args[]){  
        DrugaKlasa dk = new DrugaKlasa();  
        PrvaKlasa pk = new PrvaKlasa();  
        pk.setDrugaKlasa(dk);  
    }  
}
```



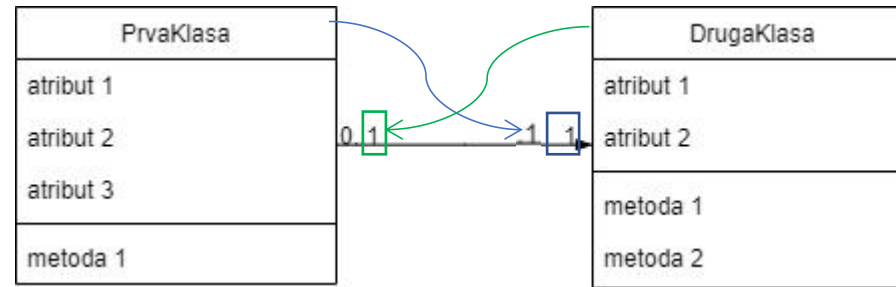
Prvi broj je 0. Objekat klase DrugaKlasa može postojati bez objekta klase PrvaKlasa.

Prvi broj je 1. Objekat klase PrvaKlasa ne može postojati bez objekta klase DrugaKlasa.

# Atribut u klasi **složenog** tipa

Kardinalitet za datu klasu  
čitamo sa suprotne strane

- Veza između klasa
  - Jednostruka veza



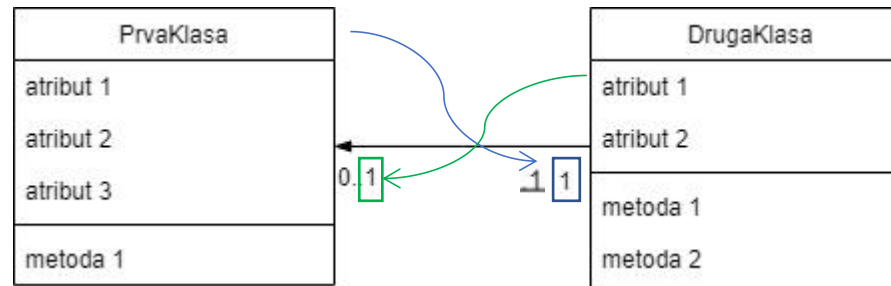
```
class PrvaKlasa{  
    DrugaKlasa referenca1;  
}
```

Zbog jednosmerne veze u klasi  
DrugaKlasa ne pravimo referencu ka  
klasi PrvaKlasa.

# Atribut u klasi **složenog** tipa

Kardinalitet za datu klasu  
čitamo sa suprotne strane

- Veza između klasa
  - Jednostruka veza



Zbog jednosmerne veze u klasi  
PrvaKlasa ne pravimo referencu ka  
klasi DrugaKlasa.

```
class DrugaKlasa{  
    PrvaKlasa referenca2;  
}
```

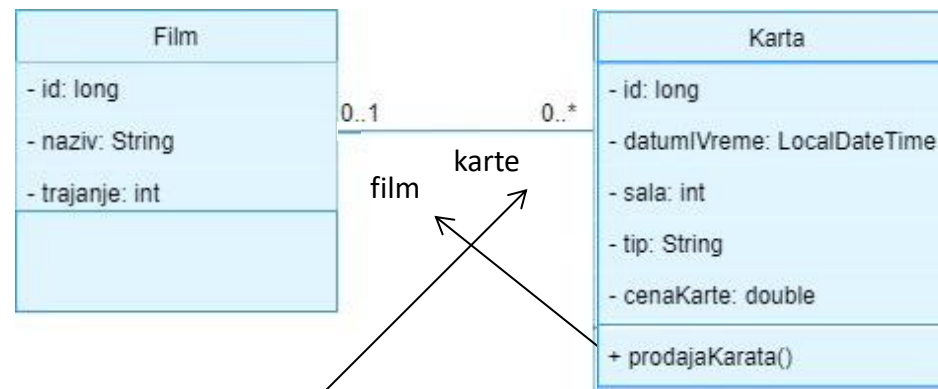
# Atribut u klasi **složenog** tipa

- Klase mogu biti opisane tekstom ili klasnim dijagramom.
- Atributi koji nisu primitivni tipovi mogu biti referenca na drugu klasu ili skup (kolekcija) referenci ka drugoj klasi.
- U nastavku su 3 moguća načina za povezivanje 2 klase objašnjene tekstom, klasnim dijagramom i kodom.



# Atribut u klasi **složenog** tipa

- Modifikacija primera Bioskop aplikacije.
  - Proširiti model podataka tako da se obuhvati entitet film. Film je opisan id-jem, nazivom i trajanjem.
  - Za svaku kartu je poznat film, i za svaki film su poznate karte.



*role name karte koji će postati naziv atributa*

*role name film koji će postati naziv atributa*

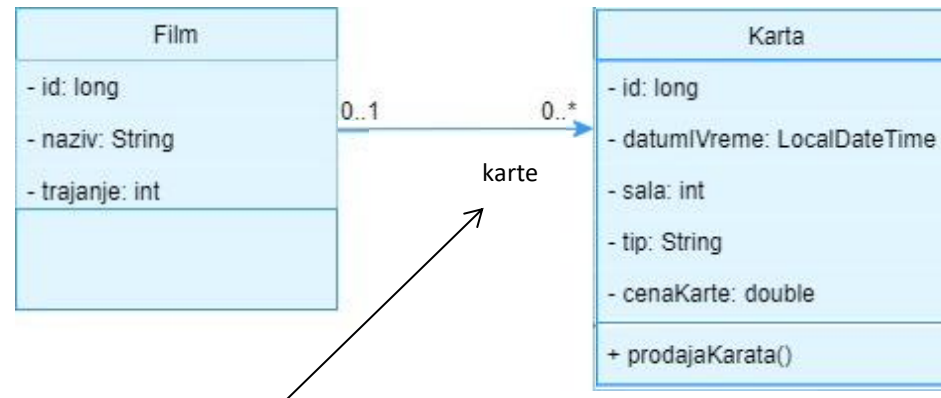
# Atribut u klasi **složenog** tipa

```
class Karta{  
    long id;  
    String datumVreme;  
    int sala;  
    String tip;  
    double cenaKarte;  
    Film film;  
    void prodajaKarata(){  
        System.out.println("Počela  
        je prodaja karata");  
    }  
}
```

```
class Film{  
    long id;  
    String naziv;  
    int trajanje;  
    ArrayList<Karta> karte;  
}
```

# Atribut u klasi **složenog** tipa

- Modifikacija primera Bioskop aplikacije.
  - Proširiti model podataka tako da se obuhvati entitet film. Film je opisan id-jem, nazivom i trajanjem.
  - Za svaki film su poznate karte.



*role name karte koji će postati naziv atributa*

# Atribut u klasi **složenog** tipa

```
class Karta{  
    long id;  
    String datumVreme;  
    int sala;  
    String tip;  
    double cenaKarte;  
    void prodajaKarata(){  
        System.out.println("Počela  
        je prodaja karata");  
    }  
}
```

```
class Film{  
    long id;  
    String naziv;  
    int trajanje;  
    ArrayList<Karta> karte;  
}
```

# Atribut u klasi **složenog** tipa (Primer u projektu)

- Modifikacija primera Bioskop aplikacije.
  - Proširiti model podataka tako da se obuhvati entitet film. Film je opisan id-jem, nazivom i trajanjem.
  - Za svaku kartu je poznat film.



*role name film koji će postati naziv atributa*

# Atribut u klasi **složenog** tipa(Primer u projektu)

```
class Karte{  
    long id;  
    String datumVreme;  
    int sala;  
    String tip;  
    double cenaKarte;  
    Film film;  
    void prodajaKarata(){  
        System.out.println("Počela je prodaja karata");  
    }  
}
```

```
class Film{  
    long id;  
    String naziv;  
    int trajanje;  
}
```

# Smernice modelovanja:

- **PODSEĆANJE SMERNICA ZA MODELOVANJE:**
  - Ako u tekstu zadatka identifikujemo varijaciju reči **više** znamo da možemo imati atribut koji će biti **Set/Lista/Mapa** i koji će sadržati reference ka **više** objekata.
  - Ako u tekstu zadatka identifikujemo reči **jedan** znamo da možemo imati atribut koji će sadržati referencu ka **jednom** objektu.

# Inicijalizacija objekta

- Ako želimo posebnu akciju prilikom kreiranja objekta neke klase, napravićemo konstruktor.
- Konstruktor je posebna metoda koja konstruiše objekat klase.
- Konstruktor se automatski poziva prilikom kreiranja objekta.
- Ukoliko ne postoji napisan ni jedan konstruktor, Java kompajler će sama generisati podrazumevani (*default*-ni).

**Karta p = new Karta();**

- Ima obavezno isto ime kao i klasa i nema povratni tip



# Konstruktori klase

- Ako ne napravimo konstruktor, kompajler će sam napraviti default konstruktor, koji ništa ne radi.
- U konstruktoru inicijalizujemo attribute koji bi trebalo da su inicijalizovani.
- Konstruktor može primiti i parametre.
- Kako je konstruktor metoda klase možemo da napravimo proizvoljan broj konstruktora sve dok se oni razlikuju po broju i tipu parametara.
- Ukoliko se u klasi napiše barem jedan konstruktor, tada podrazumevani konstruktor više ne postoji.

# Konstruktori klase - primeri konstruktora

```
class Karta{
    private long id;
    private String datumVreme;
    private int sala;
    private String tip;
    private double cenaKarte;
    Film film;

    public Karta(long id, String datumVreme, Film film, int sala, String tip, double cenaKarte) {
        this.id = id;
        this.datumVreme = datumVreme;
        this.sala = sala;
        this.tip = tip;
        this.cenaKarte = cenaKarte;
        this.film = film;
    } //konstruktor sa 6 parametara, sa id-jem

    public Karta(String datumVreme, Film film, int sala, String tip, double cenaKarte) { //konstruktor sa 5 parametra, bez id-ja
        this(0, datumVreme, film, sala, tip, cenaKarte);
    }

    public Karta() { this(0, "", null, 0, "", 0.0); } //konstruktor sa defaultnim vrednostima
}
```

# Objekti - pozivanje konstruktora

- Primer kreiranje Objekta pozivanjem različitih konstruktora

```
class Bioskop {  
    public static void main(String args[]){  
        Karta a = new Karta ();  
        Film f= new Film();  
        Karta b = new Karta (1, "12.03.2022.", f, 1, "2D",209.0);  
        Karta c = new Karta ("13.03.2022.", f, 2, "3D",309.0);  
    }  
}
```

# Literal Null i This


- Ako želimo da inicijalizujemo referencu tako da ona ne ukazuje ni na jedan objekat, onda takvoj promenljivoj dodeljujemo **null** vrednost, odn. **null** literal:  
    **Karta k = null;**  
    **k.film = null;**
- **This** ključna reč je promenljiva koja se odnosi na trenutni objekat metode ili konstruktora. Glavna svrha korišćenja ove ključne reči u Javi je uklanjanje zabune između atributa klase i parametara metode koji imaju ista imena.

# Ključna reč this

- Ključna reč **this** se koristi u entitet klasi (klasa koja ne sadrži main funkciju). Primeri korišćenja:

```
public class Bioskop {  
    public static void main(String[] args) {  
        Karta karta = new Karta();           ← 1  
        karta.prodajaKarata();               ← 2  
        Film film = new Film();  
        Karta karta1 = new Karta ("1.12.2022.",film , 1,"2D", 892.09); ← 3  
    }  
}
```

# Ključna reč this

- **Karta karta = new Karta();**  1
  - Ovom linijom je pozvan default-ni konstruktor iz klase Karta:
    - default-nim konstruktorom postavljamo default-ne vrednosti atributa. Možemo na 2 načina postaviti osnovne vrednosti tipova:


1. Pozivanjem konstruktora sa svim vrednostima (primer u projektu):

```
public Karta() {  
    this(0, "", null, 0, "", 0.0); }  
// Postavljamo vrednosti kreiranog objekta karta (sa prethodnog slajda) pozivajući konstruktor sa  
// svim parametrima
```

```
public Karta(String datumIVreme, Film film, int sala, String tip, double cenaKarte);
```

2. Postavljanjem svakog atributa u datom konstruktoru:

```
public Karta() {  
    this.id = 0;  
    this.datumIVreme = "";  
    this.film = null;  
    this.sala = 0;  
    this.tip = "";  
    this.cenaKarte = 0.0; }  
// Postavljamo vrednosti kreiranog objekta karta (sa prethodnog slajda).
```



# This ključna reč

- `karta.prodajaKarata();` ← 2
  - Metode se mogu pozivati samo nad objektima.
- `Karta karta1 = new Karta("1.12.2022.",film , 1,"2D", 892.09);` ← 3

- Poziva se konstruktor:

```
public Karta(String datumIVreme, Film film, int sala, String tip, double cenaKarte){  
    this.id = 0;  
    this.datumIVreme =datumIVreme; // Postavljamo vrednosti kreiranog objekta karta1  
    this.film = film;  
    this.sala = sala;  
    this.tip = tip;  
    this.cenaKarte = cenaKarte;  
}
```

ili drugi način(u primeru):

```
public Karta(String datumIVreme, Film film, int sala, String tip, double cenaKarte){  
    this(0, datumIVreme, film, sala,tip, cenaKarte);  
}
```

Oba konstruktora postavljaju vrednosti svih parametara sem id-ja.

Koristite pristup koji Vam je lakši, pokazujemo oba zbog ključne reči this.

# This ključna reč

- **karta.prodajaKarata();**

```
public void prodajaKarata(){  
    this.sala = 2;  
    ...  
}
```

Gde se promenio broj sale,  
nad kojim objektom?



# This ključna reč

- **karta**.prodajaKarata();

```
public void prodajaKarata(){  
    this.sala = 2;  
    ...  
}
```

Nad objektom, nad kojim  
smo pozvali metodu  
prodajaKarata.

# Modifikatori pristupa

- Ponekad je potrebno obezbediti kontrolisan pristup atributima, kako za čitanje, tako i za pisanje. Tada se koriste se modifikatori pristupa
  - **public** – vidljiv za sve klase
  - **protected** – vidljiv samo za klase naslednice i klase iz istog paketa
  - **private** – vidljiv samo unutar svoje klase
  - **nespecificiran** (package-private) – vidljiv samo za klase iz istog paketa (direktorijuma, foldera)
- Modifikatori pristupa se navode ispred definicija klasa, metoda i atributa.

# Get i Set - Enkapsulacija

- Kada je potrebno obezbediti kontrolisan pristup atributima (čitanje i izmena vrednosti) koriste se *getters* i *setters* metode.

```
public class Film{  
    private String naziv;  
  
    public String getNaziv() {  
        return naziv;  
    }  
    public void setNaziv(String naziv) {  
        this.naziv= naziv;  
    }  
}
```

# Modifikatori pristupa

- Kada atributima i metodama odredimo i napišemo modifikatore pristupa, dobija se klasa kojoj je omogućen **kontrolisani** pristup iz drugih klasa i programa.
- Klase mogu međusobno da komuniciraju bez znanja o međusobnoj implementaciji njihovih metoda, npr. da li pozvana javna metoda u svom telu poziva i neke druge skrivene metode ili da ona koristi neke skrivene attribute, itd.
- **Detalji implementacije su skriveni**, tj. **enkapsulirani** unutar klase. Druga klasa vidi prvu klasu samo kroz metode kojima ona može da pristupi!!!

# Get i Set - Enkapsulacija

- Kombinacija nevidljivog atributa i vidljivih get i set metoda naziva se svojstvo (*property*).
- Ovim je omogućeno da se čitanje vrednosti svojstva samo sprovodi kroz njegov *getter*, a izmena samo kroz *setter*.
- Ako izostavimo *setter*, dobijamo *read only* svojstvo.

# Modifikatori pristupa - primer

- Primer rada sa public atributima:

## Klasa

```
public class Proizvod {  
    public long id;  
    public String naziv;  
    public double cena;  
    public int brojKopija;  
}
```

## Primer pristupa

```
Proizvod p = new Proizvod ();  
p.naziv = "Novi naziv";
```

# Modifikatori pristupa - primer

- Primer rada sa protected atributima:

## Klasa

```
public class Proizvod {  
    protected long id;  
    protected String naziv;  
    protected double cena;  
    protected int brojKopija;  
}
```

## Primer pristupa

```
Proizvod p = new Proizvod ();  
p.naziv = "Novi naziv";
```

# Modifikatori pristupa - primer

- Primer rada sa nespecificiranim atributima:

## Klasa

```
public class Proizvod {  
    long id;  
    String naziv;  
    double cena;  
    int brojKopija;  
}
```

## Primer pristupa

```
Proizvod p = new Proizvod ();  
p.naziv = "Novi naziv";
```



# Modifikatori pristupa - primer

- Primer rada sa private atributima:

## Klasa

```
public class Proizvod {  
    private long id;  
    private String naziv;  
    private double cena;  
    private int brojKopija;  
}
```

## Primer pristupa

```
Proizvod p = new Proizvod ();  
p.setNaziv("Novi naziv");
```

# Funkcije (Metode)

- Izdvojeni skup programskog koda koji se može pozvati (izvršiti) u bilo kom trenutku u programu
- Dekompozicijom programa u manje izdvojene celine (funkcijama) eliminiše se potreba za ponavljanjem koda. Prednost:
  - Kod organizovaniji
  - Kod čitljiviji
- Najlakše objasniti ako se posmatraju kao podprogrami (podalgoritmi) specifične namene

# Funkcije (Metode)

```
public class ProbnaKlasa{
```

modifikator

povratni tip

naziv metode

parametri metode

public

int

primerMetode

(int a, int b)

{

int c = a + b;  
return c;

}

telo metode

}

}

# Funkcije (Metode)

- **NAPOMENA: NIKADA NEĆEMO KREIRATI FUNKCIJU JEDNU UNUTAR DRUGE**
- Metoda u javi može imati **jedan** povratni tip. Povratni tip može biti:
  - primitivni tip
  - referenca na objekat (bilo šta kreirano ključnom rečju **new**)
  - nema povratnog tipa (void)
- Metoda u javi može imati **proizvoljan** broj ulaznih parametara. Ulazni tip može biti:
  - primitivni tip
  - referenca na objekat (bilo šta kreirano ključnom rečju **new**)

# Funkcije (Metode)

- Funkcija se može pozvati u bilo kom trenutku.
- Primer pozivanja funkcije:

```
public static void main(String args[])  
{  
    primerMetode(3, 4);  
}
```

naziv metode

pozivajući parametri (ukoliko ih ima)

U Javi se svaka linija koda završava ;

# Funkcije (Metode)

- U klasi može da postoji više funkcija sa istim imenom (method overloading)
- Povratna vrednost funkcije:

```
public int primerMetode (int a, int b)
{
    int c = a + b;
    return c;
}
```

Metoda vraća vrednost naredbom **RETURN**

# Funkcije (Metode)

- Povratni tip zaglavlja metode i povratna vrednost metode:

```
public int primerMetode (int a, int b)
{
    int c = a + b;
    return c;
}
```



Tip povratne vrednosti iz tela funkcije se **MORA** poklopiti sa tipom povratne vrednosti zaglavlja funkcije

- **NAPOMENA 1:** Konverzija sa šireg na uži tip **NIJE** dozvoljena. Konverzija sa užeg na širi tip **JE** dozvoljena.
- **NAPOMENA 2:** **NIKADA** nećemo pisati kod nakon return naredbe.

# Modifikatori pristupa - rezime

- Klase i metode će imati public modifikator pristupa.
- Atributi će imati private modifikator pristupa sa get/set metodama.



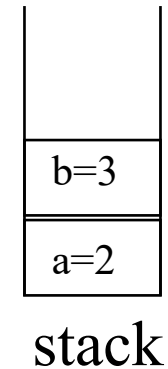
# Čuvanje promenljivih - rad sa stack memorijom

- Sve promenljive koje se koriste u programu se čuvaju u delu memorije označen kao ***stack***.
- 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.

# Čuvanje promenljivih - rad sa stack-om

- Prilikom smeštanja vrednosti na steku zauzima se (alocira se) memorija u kojoj će se smestiti vrednost primitivnog tipa. Zauzimanje memorije zavisi od veličine datog tipa (int - 32 bita, float -32 bita,...)

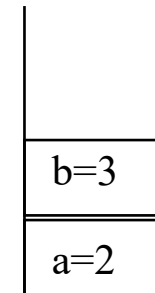
```
public static void main(String[] args) {  
    int a = 2;  
    int b = 3; ←  
    int c = 5;  
}
```



# Čuvanje promenljivih - rad sa stack-om

- 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);  
}
```



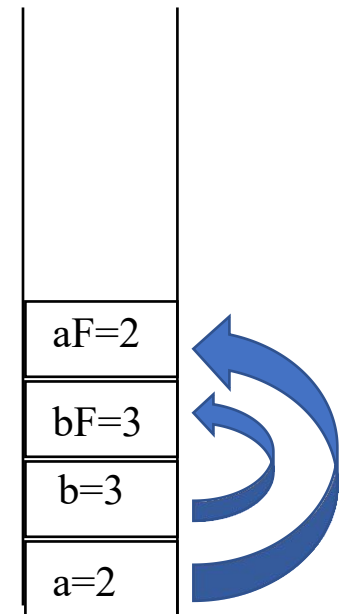
stack



# Čuvanje promenljivih - rad sa stack-om


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

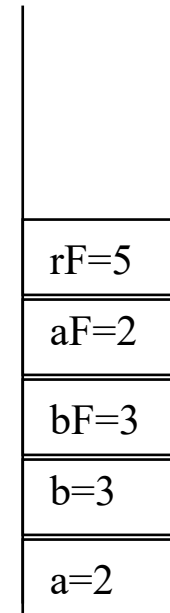
```
public static int obradiZbir (int bF, int aF) {  
    int rF = aF + bF;  
    return rF;  
}  
public static void main(String[] args) {  
    int a = 2;  
    int b = 3;  
    int c = obradiZbir(b, a);  
}
```



# Čuvanje promenljivih - rad sa stack-om

- 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);  
}
```

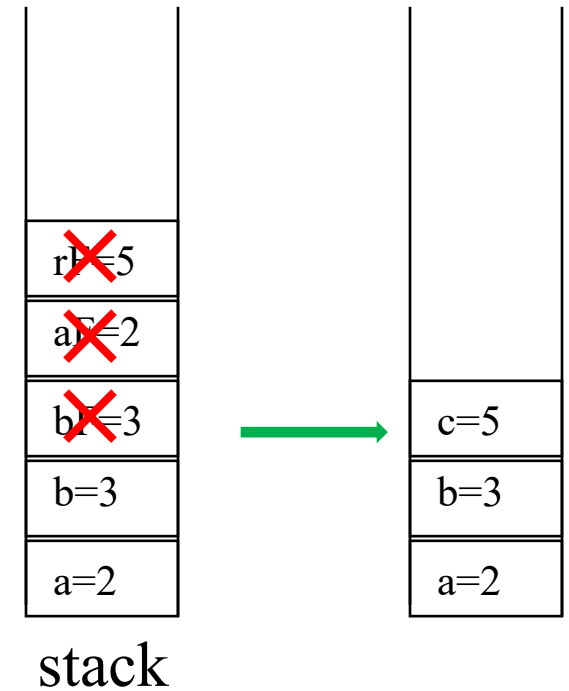


stack

# Čuvanje promenljivih - rad sa stack-om

- 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);  
}
```

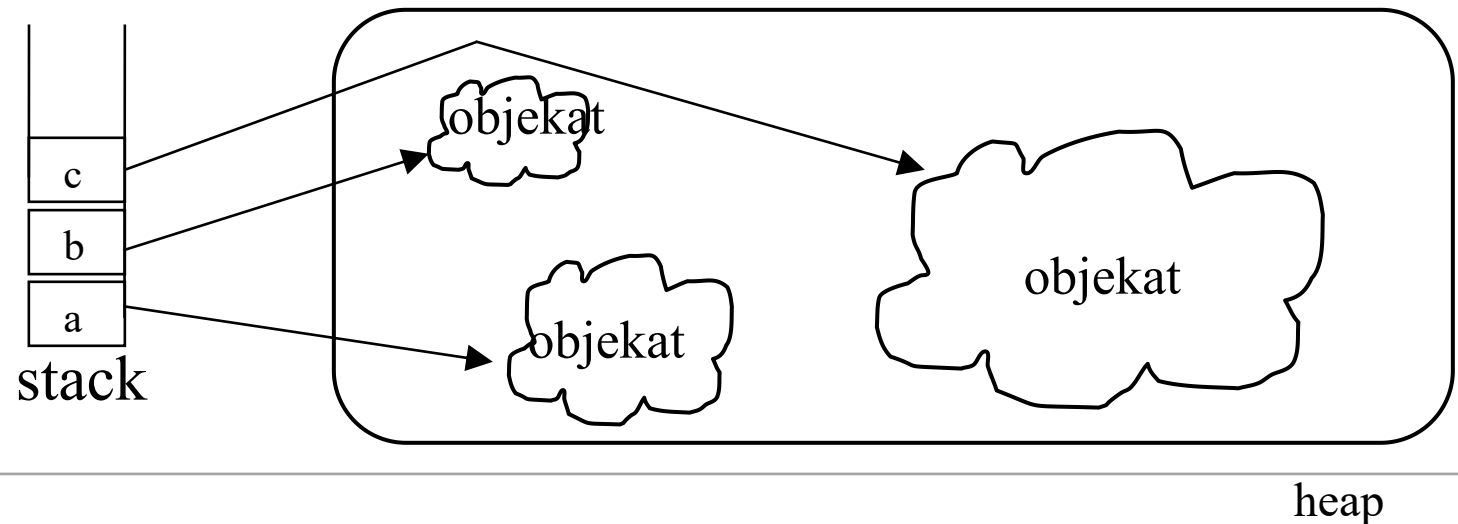


# Heap i Objekti

- **Heap** ili dinamička memorija je memorija gde se smeštaju dinamički alocirane vrednosti. Objekti su dinamički alocirane vrednosti.
- Osnovna klasa za sve objekte u Javi je klasa **Object**
- Objekti se kreiraju ključnom rečju **new**.

# Objekti

- 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**

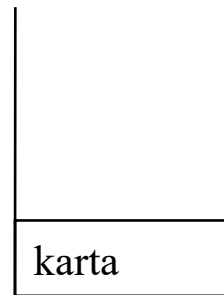




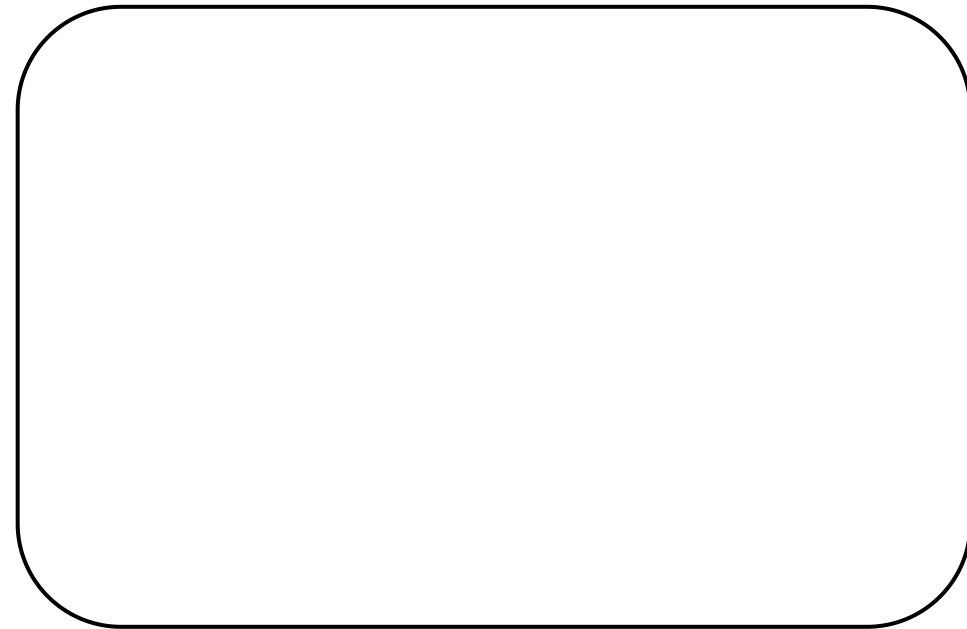
# Čuvanje promenljivih - rad sa heap-om

- Na steku se samo deklariše promenljiva:

Karta karta;



stack

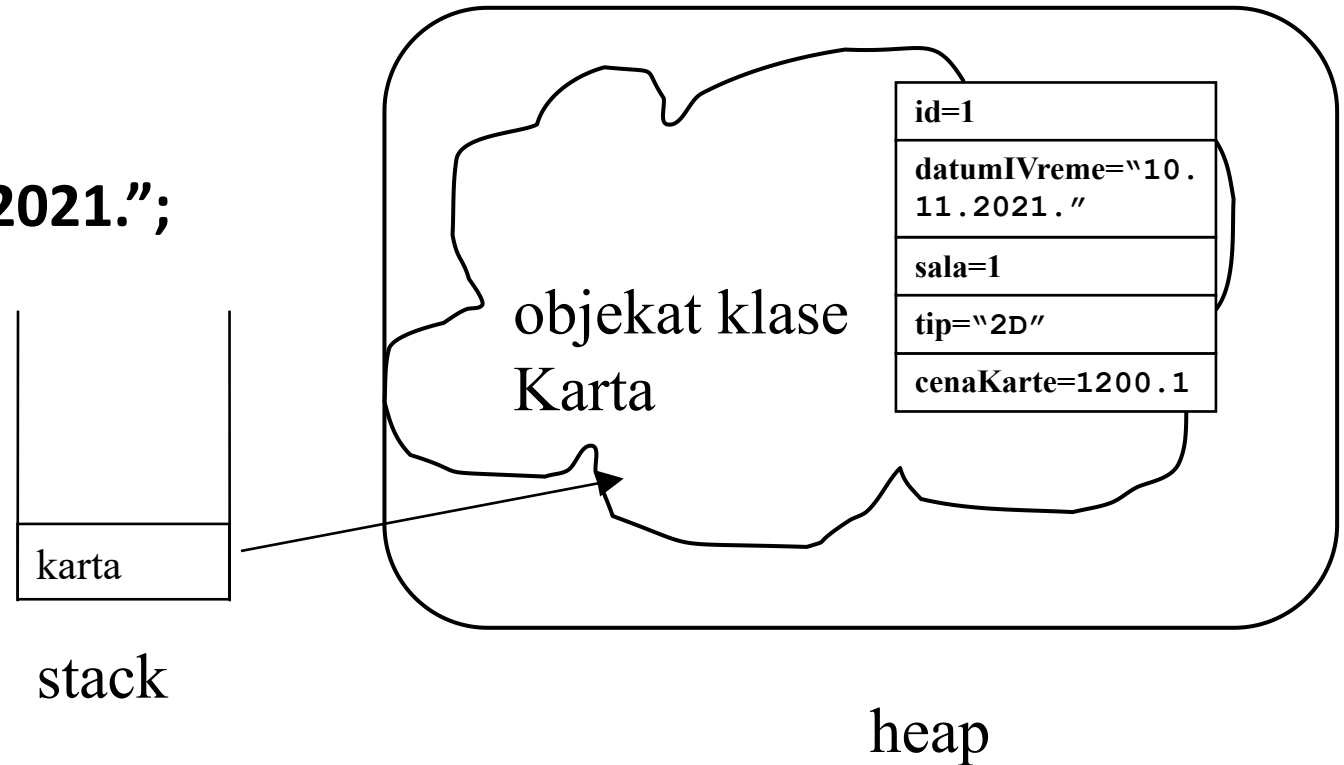


heap

# Čuvanje promenljivih - rad sa heap-om

- Dodela vrednosti određenom atributu objekta i pozivanje metode objekta.

```
Karta karta;  
karta= new Karta ();  
karta.id = 1;  
karta.datumIVreme = "10.11.2021.";  
karta.sala = 1;  
karta.tip = "2D";  
karta.cenaKarte =1200.1;
```



# Objekti - atributi koji nisu primitivni tipovi

Primer kreiranja (instanciranja) objekta:

```
class Test {  
    public static void main(String args[]) {  
        Film film = new Film();  
        film.id = 1;  
        film.naziv = "Juzni vetar";  
        film.trajanje = 2; ← 1  
        Karta k;  
        k = new Karta();  
        p.id = 1;  
        k.datumIVreme = "10.11.2021.";  
        k.sala = 1;  
        k.tip = "2D";  
        k.cenaKarte = 1200.1;  
        k.prodajaKarata();  
        k.film=film; ← 3  
        k.film.naziv="Nacionalna klasa"; ← 4  
    }  
}
```

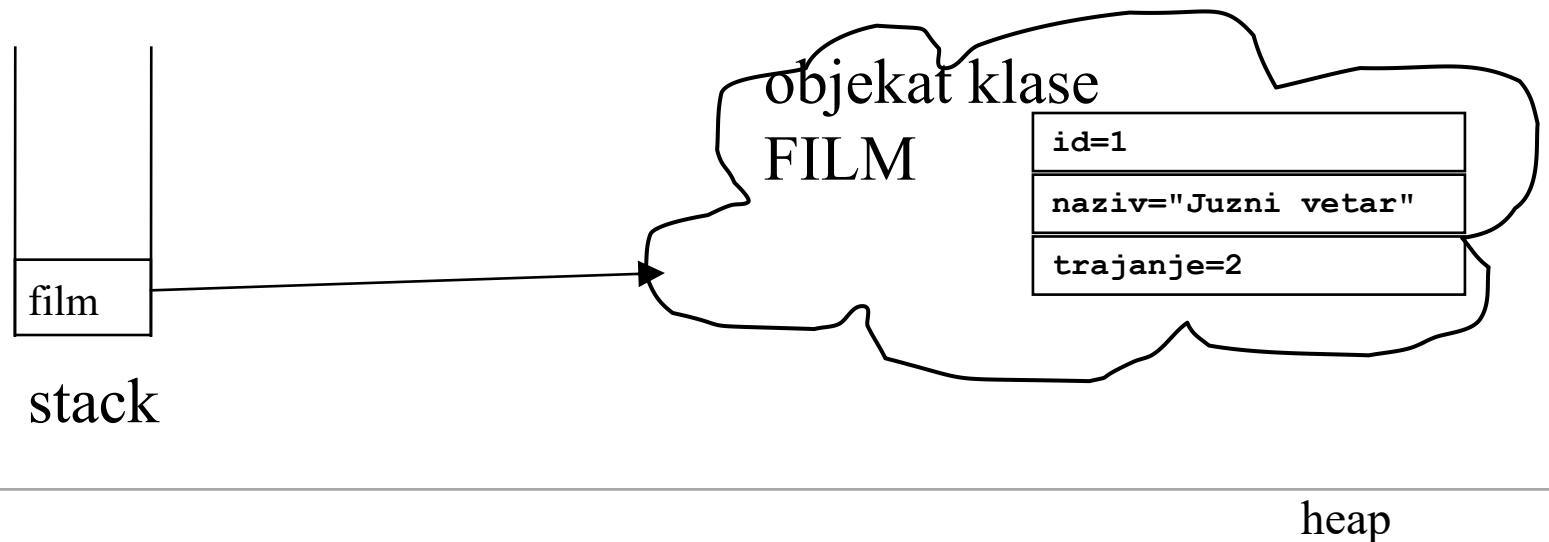
The diagram illustrates the creation of objects and the assignment of non-primitive attributes. Four numbered arrows point to specific lines of code:

- 1: Points to `film.trajanje = 2;`
- 2: Points to `k.prodajaKarata();`
- 3: Points to `k.film=film;`
- 4: Points to `k.film.naziv="Nacionalna klasa";`

# Objekti - preslikavanje na Heap memoriju

- Kod izvršen do ← 1

```
Film film = new Film();  
film.id = 1;  
film.naziv = "Juzni vetar";  
film.trajanje = 2;
```



# Objekti - preslikavanje na Heap memoriju

- Kod izvršen do



Karta k;

k = **new** Karta ();

k.id = 1;

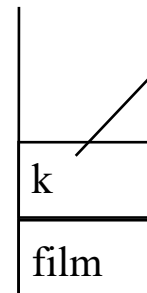
k.datumIVreme = "10.11.2021.";

k.sala = 1;

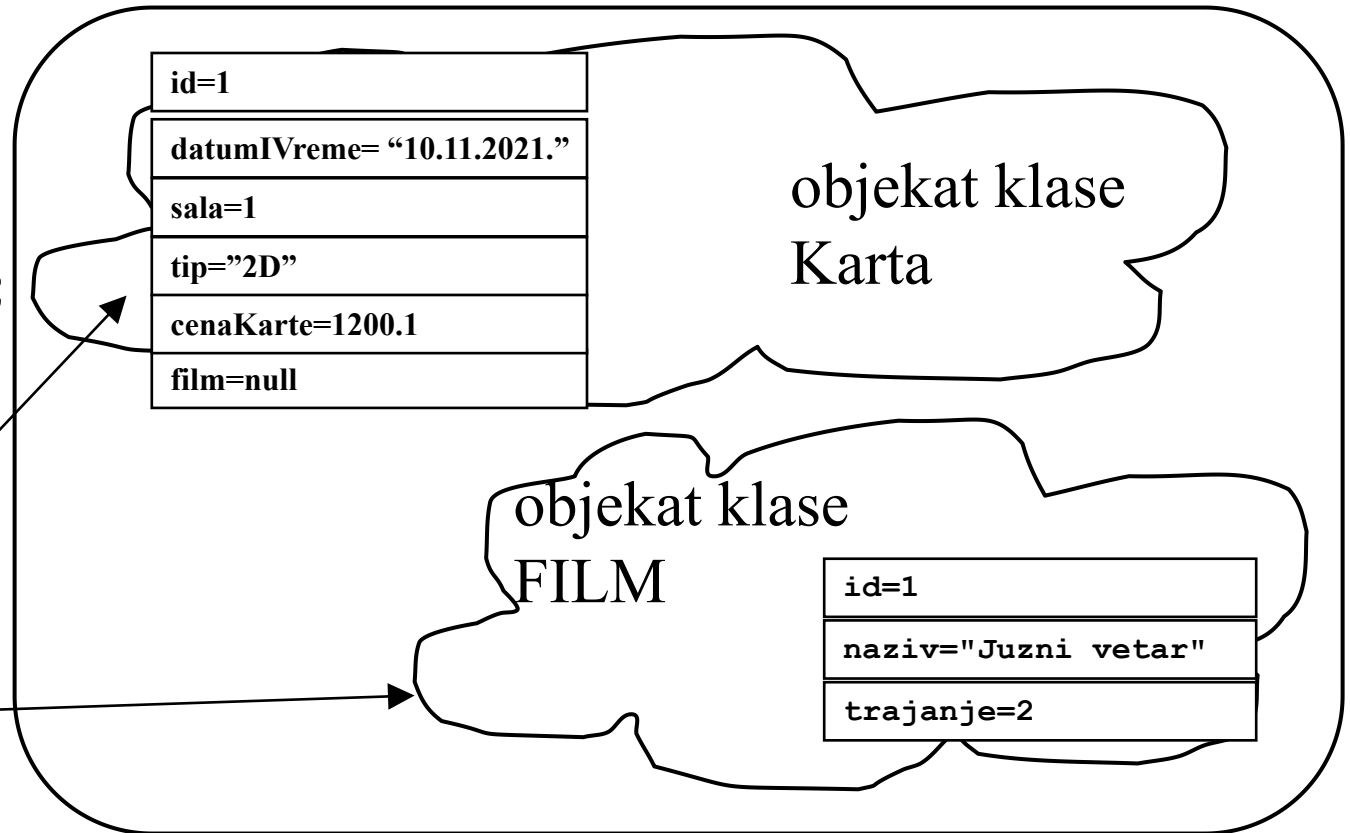
k.tip = "2D";

k.cenaKarte = 1200.1;

k.prodajaKarata();



stack

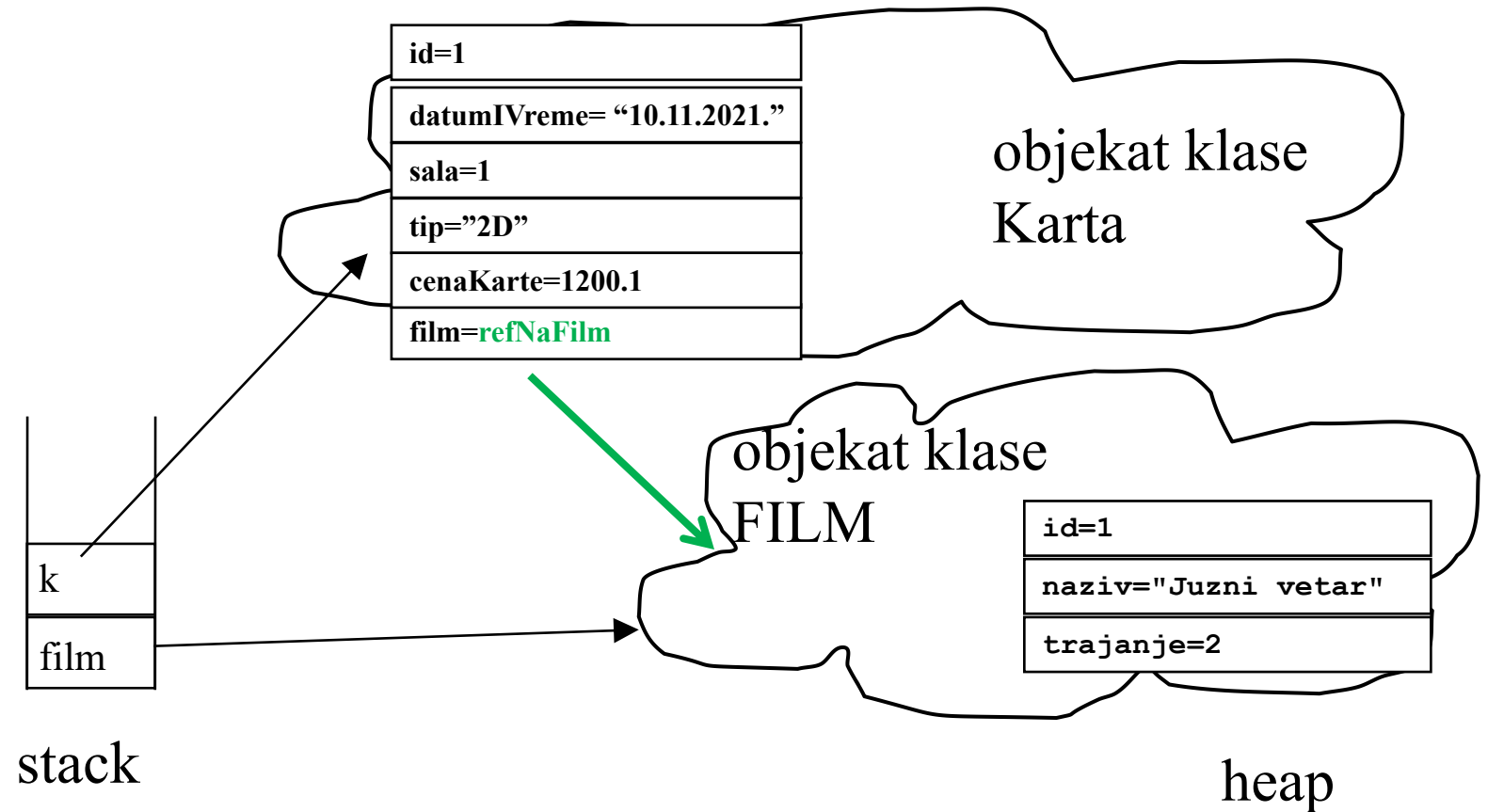


heap

# Objekti - preslikavanje na Heap memoriju

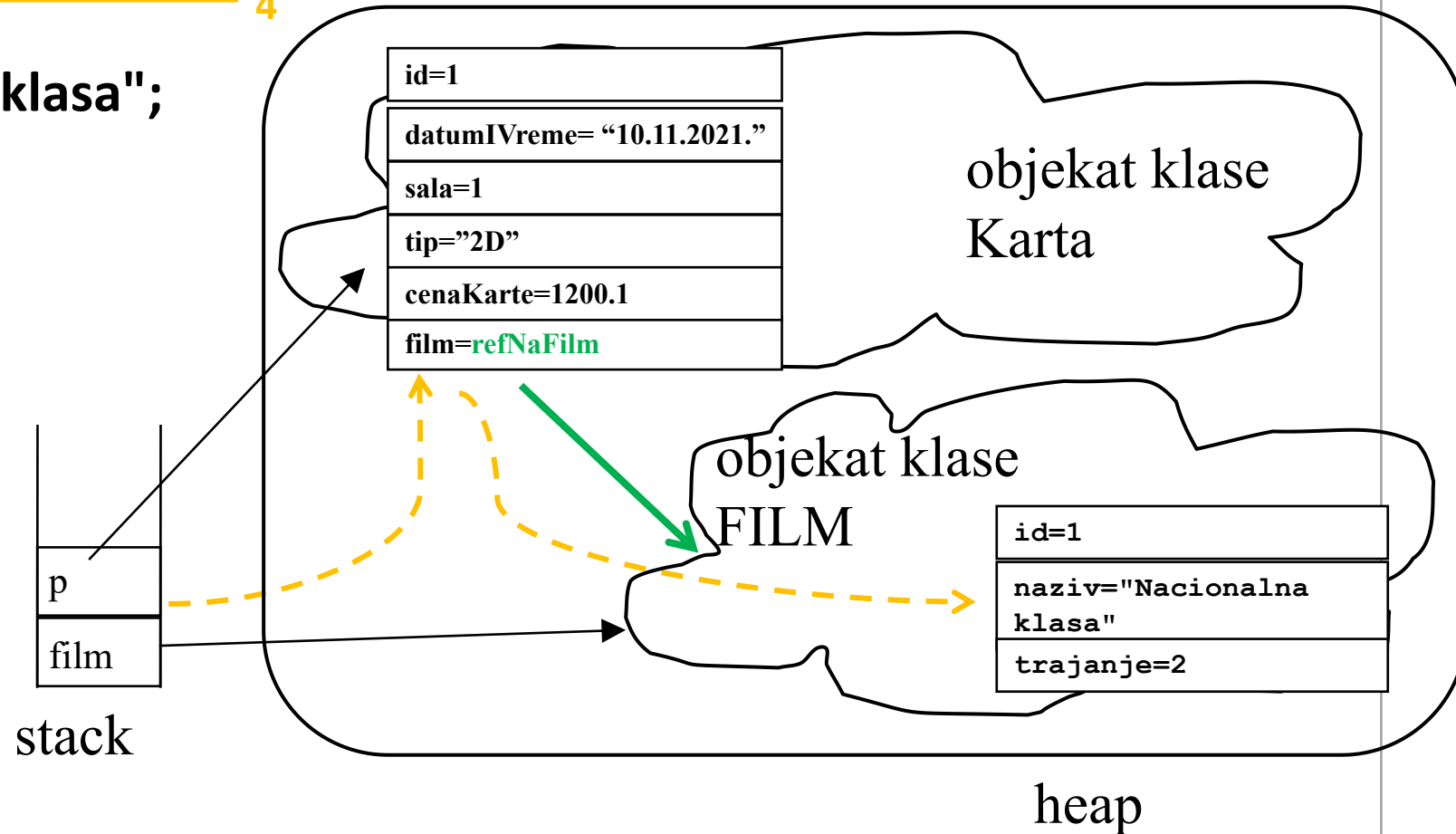
- Kod izvršen do
- **k.film=film;**

← 3



# Objekti - preslikavanje na Heap memoriju

- Kod izvršen do ← 4
- `k.film.naziv="Nacionalna klasa";`



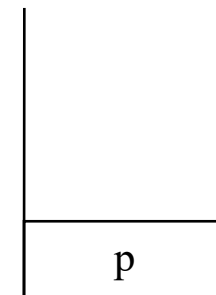
# Objekti - Referenca na objekat kao parametar metode

```
void metodaKlase(Karta kF) {  
    kF.sala = 2;  
}  
void main(args[]) {  
    Karta k = new Karta();  
    k.id = 1;  
    k.datumIVreme = "10.11.2021.";  
    k.sala = 1;  
    k.tip = "2D";  
    k.cenaKarte = 1200.1;  
    metodaKlase(k);  
}
```

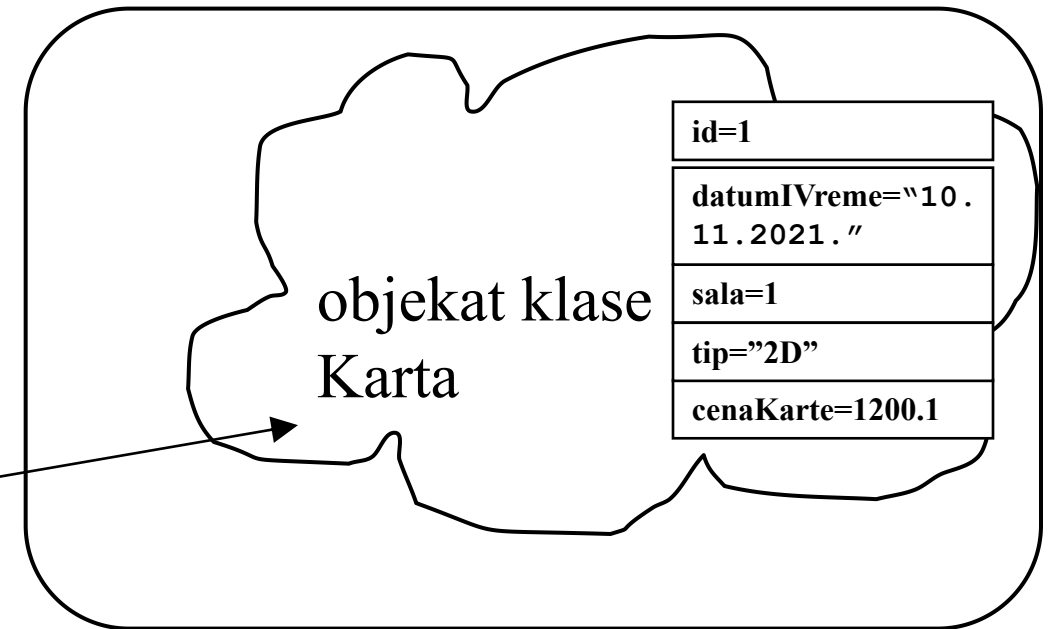


# Objekti - Referenca na objekat kao parametar metode

```
k = new Karta();  
k.id = 1;  
k.datumIVreme = "10.11.2021.";  
k.sala = 1;  
k.tip = "2D";  
k.cenaKarte = 1200.1;
```



stek

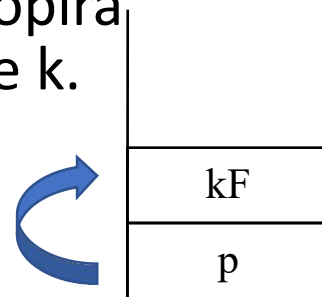


heap

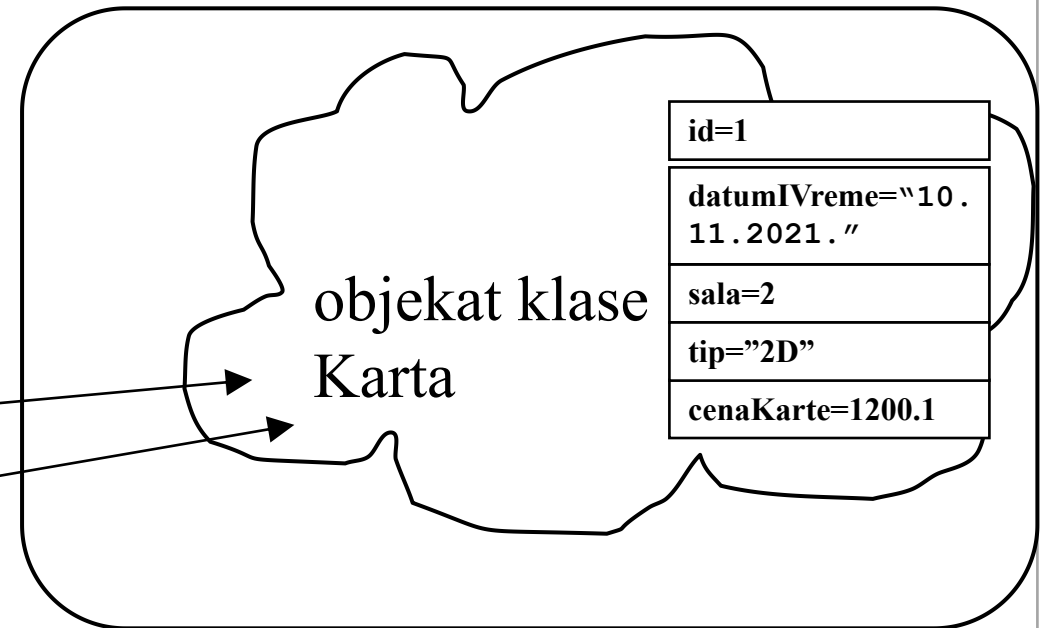
# Objekti - Referenca na objekat kao parametar metode

```
void metoda(Karta kF) {  
    kF.sala = 2;  
}  
...  
metodaKlase(k);  
...
```

Na steku prenos je strogo po vrednosti,  
u promenljivoj kF se kopira  
vrednost iz promenljive k.



stek



# Objekti - Referenca na objekat kao parametar metode

```
class MainKlasa(){  
    void metodaKlase(Karta kF) {  
        kF.sala = 2;  
    }  
    void main(args[]) {  
        Karta k = new Karta();  
        k.id = 1;  
        k.datumIVreme = "10.11.2021.";  
        k.sala = 1;  
        k.tip = "2D";  
        k.cenaKarte = 1200.1;  
        metodaKlase(p);  
        // koja je vrednost atributa sala???  
    }  
}
```

# Slanje parametara - rezime

- Slanje parametra može biti po referenci i vrednosti.
  - Ukoliko šaljete kopiju reference, direktno **se menja** originalni objekat.
  - Ukoliko šaljete kopiju vrednosti, **ne menja se** originalna vrednost.

# Zadatak

- Unutar klase Bioskop napraviti
  - funkciju *probnaFunkcija1*:
    - koja će kao ulazni parametar imati referencu na objekat klase Film.
    - u telu funkcije ćete izmeniti atribut *naziv* na proizvoljnu vrednost.
    - Ispisati vrednosti objekata:  
`System.out.println("Ispis iz funkcije probnaFunkcija1 " + film.toString()) .`
  - funkciju *probnaFunkcija2*:
    - koja će kao ulazni parametar imati numeričku (int) vrednost pod nazivom *broj*.
    - u telu funkcije ćete izmeniti ulazni parametar na proizvoljnu vrednost .
    - Ispisati vrednosti datog parametra:  
`System.out.println("Ispis iz funkcije probnaFunkcija2 " + broj) .`
- Unutar date klase Bioskop i main funkcije:
  - instancirati objekat klase Film,
  - napraviti promenljivu broj tipa int dodeliti joj vrednost 3
  - ispisati objekat klase FILM i ulazni parametar pod nazivom broj.
  - pozvati funkcije *probnaFunkcija1* i *probnaFunkcija2* sa odgovarajućim ulaznim parametrima.
  - Nakon poziva funkcija ispisati objekat klase FILM i ulazni parametar pod nazivom broj.

# DODATNI MATERIJALI

# Uništavanje Objekta

- U Javi ne postoji metoda destruktor za uništavanje objekata kao u C++.
- Možemo napisati posebnu metodu ***finalize()***, koja se poziva neposredno pre oslobađanja memorije koju je objekat zauzimao, ali opet nemamo garanciju da će metoda ikada biti pozvana.

# 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[3][]; //niz ima 3 vrste
```

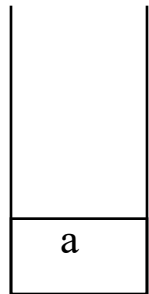
```
a[0] = new int[1]; //0 vrsta ima 1 kolona
```

```
a[1] = new int[2]; //1 vrsta ima 2 kolona
```

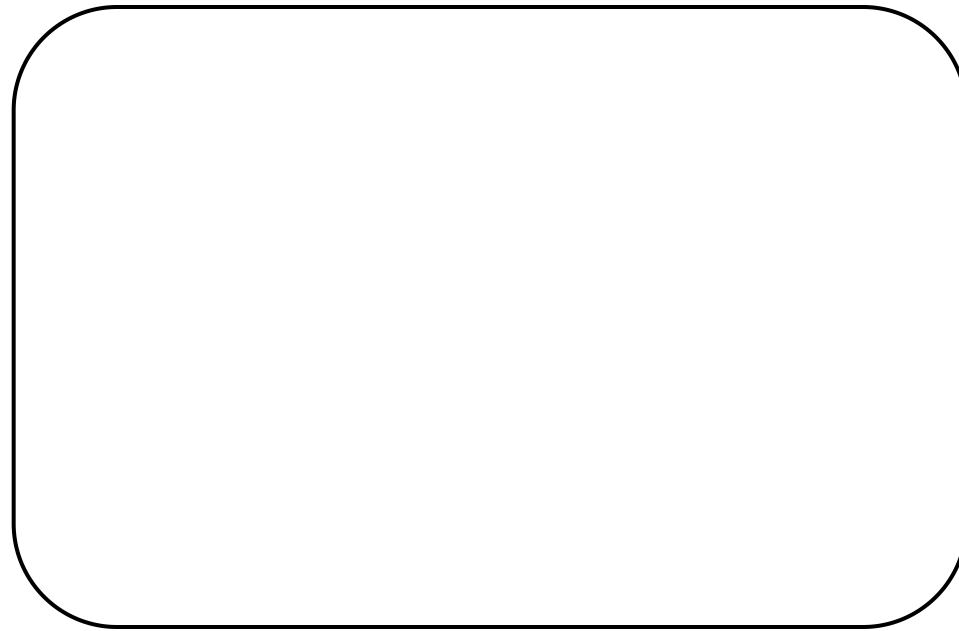


# Rad sa nizovima primitivnih tipova

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



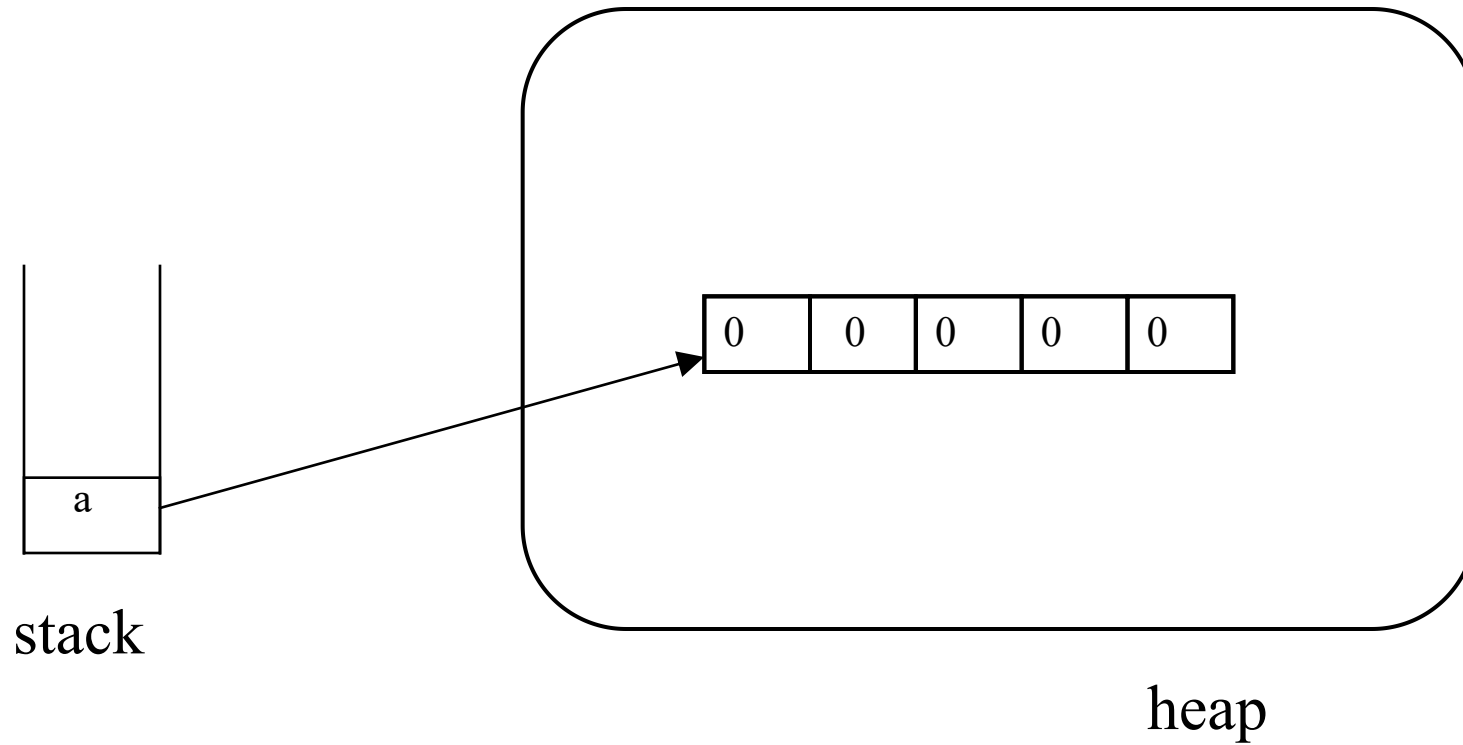
stack



heap

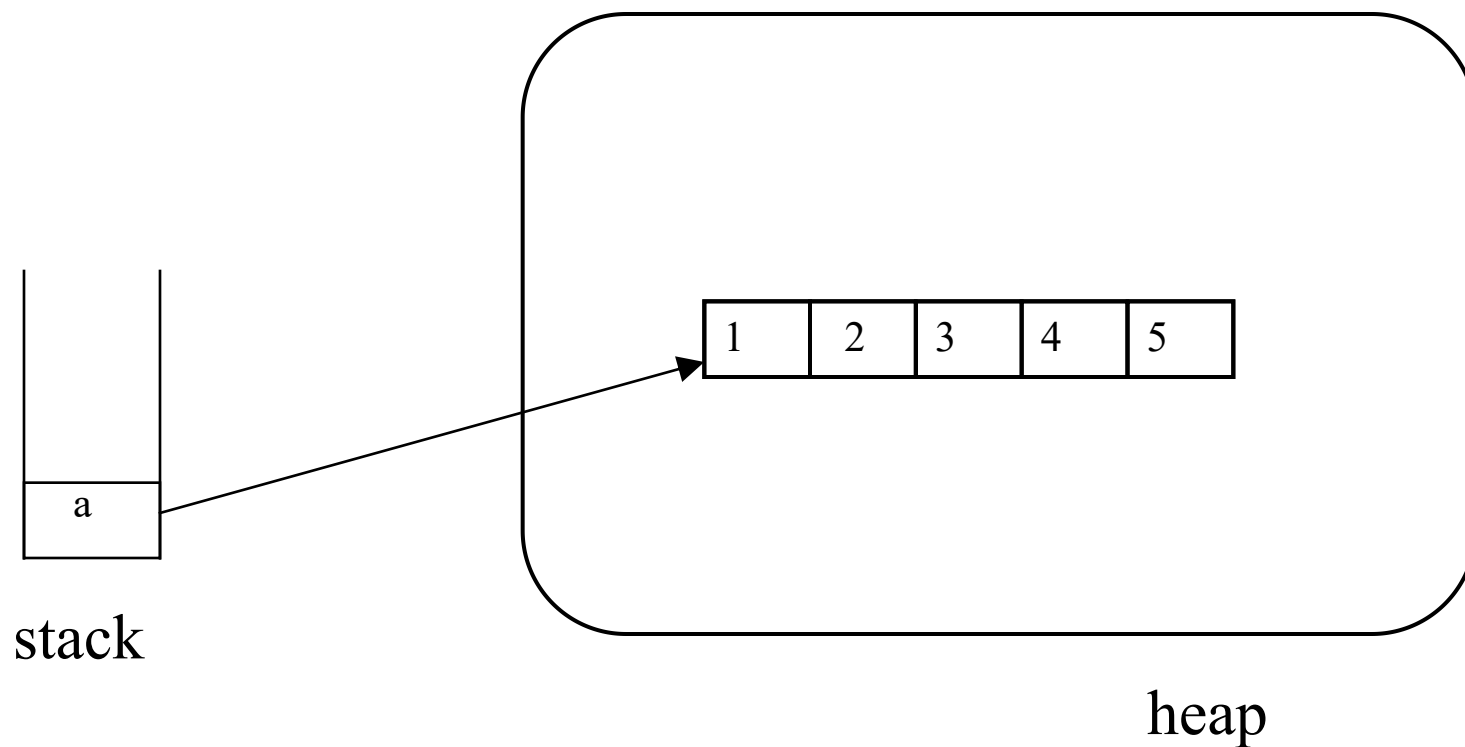
# Rad sa nizovima primitivnih tipova

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



# Rad sa nizovima primitivnih tipova

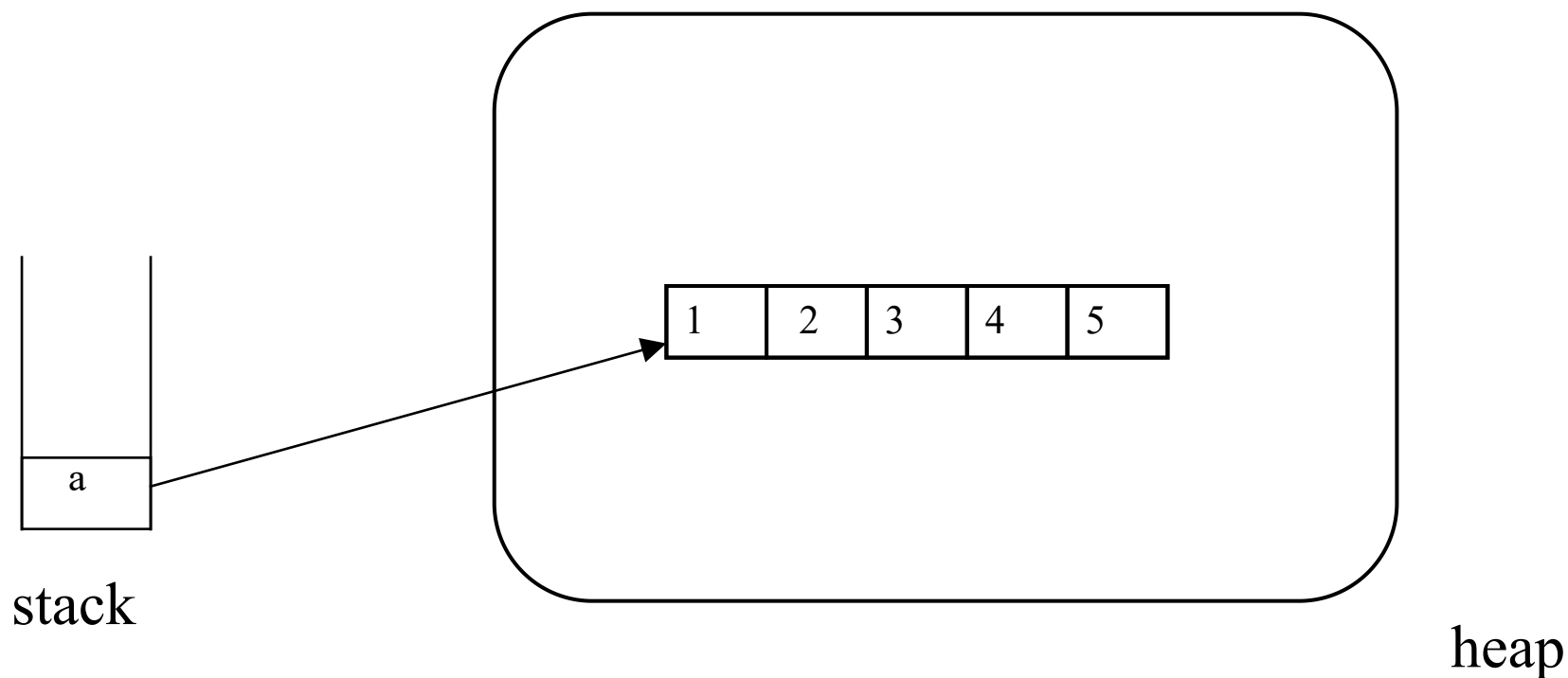
**`a[0]=1; a[1]=2; a[2]=3; a[3]=4; a[4]=5;`**



# Rad sa nizovima primitivnih tipova

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

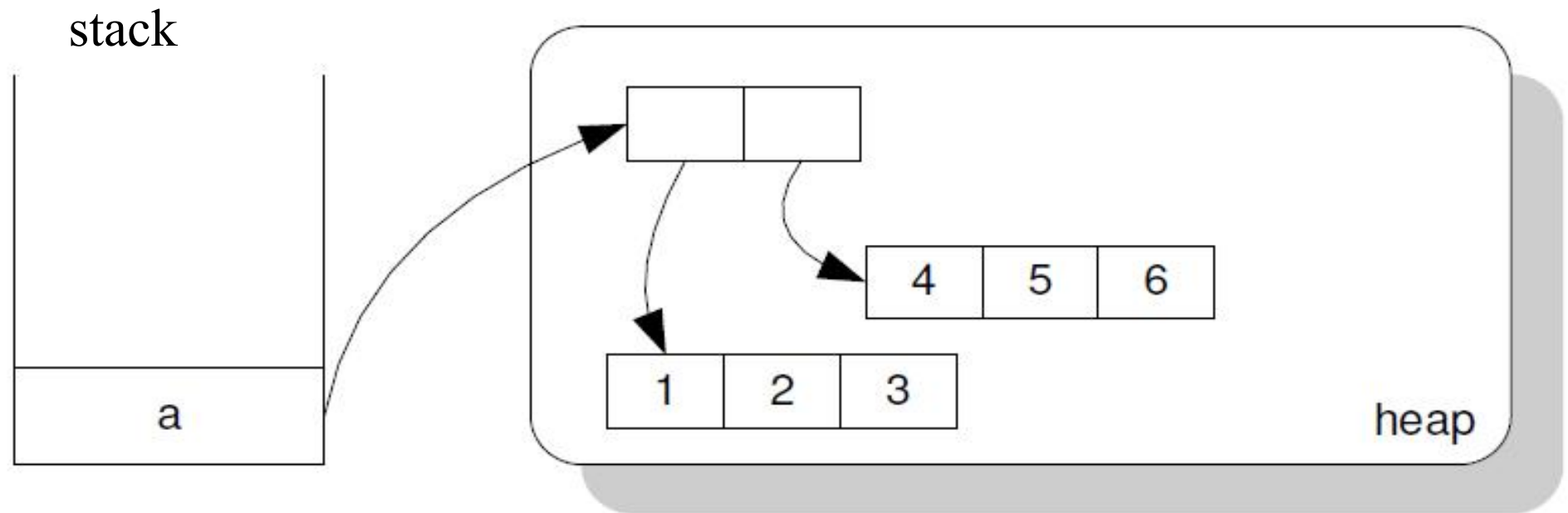
```
int a[] = { 1, 2, 3, 4, 5 };
```



# Višedimenziionalni nizovi

- U jednoj liniji kod

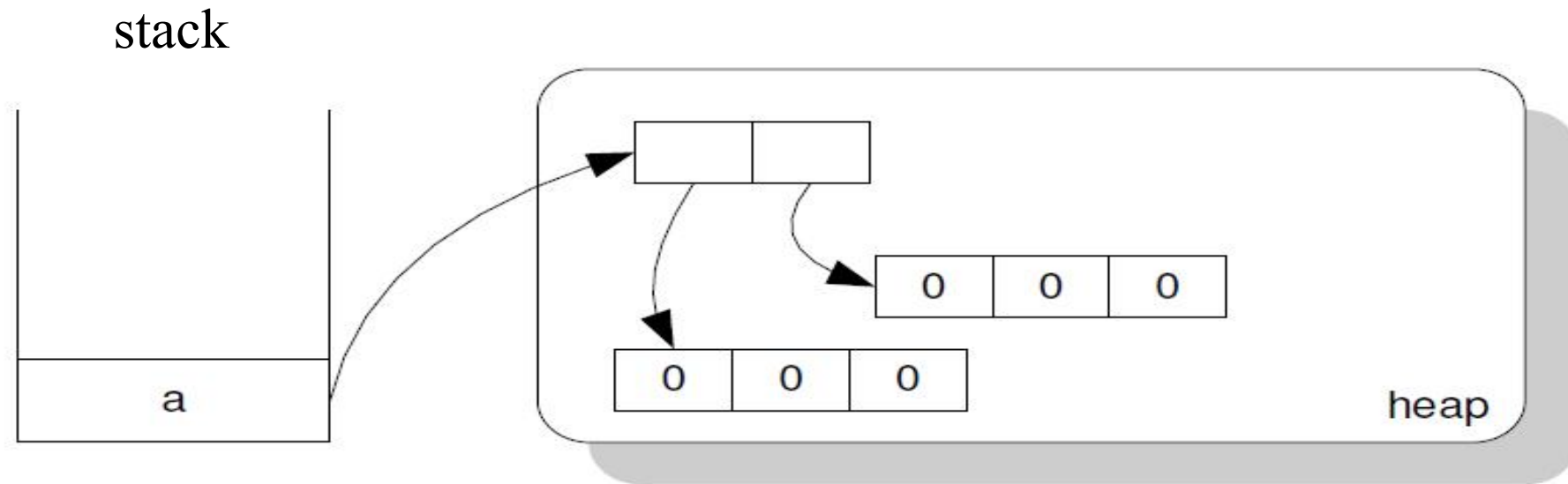
```
int[][] a = { {1, 2, 3}, {4, 5, 6} };
```



# Višedimenzionalni nizovi

- Mogu se odmah definisati sve dimenzije:

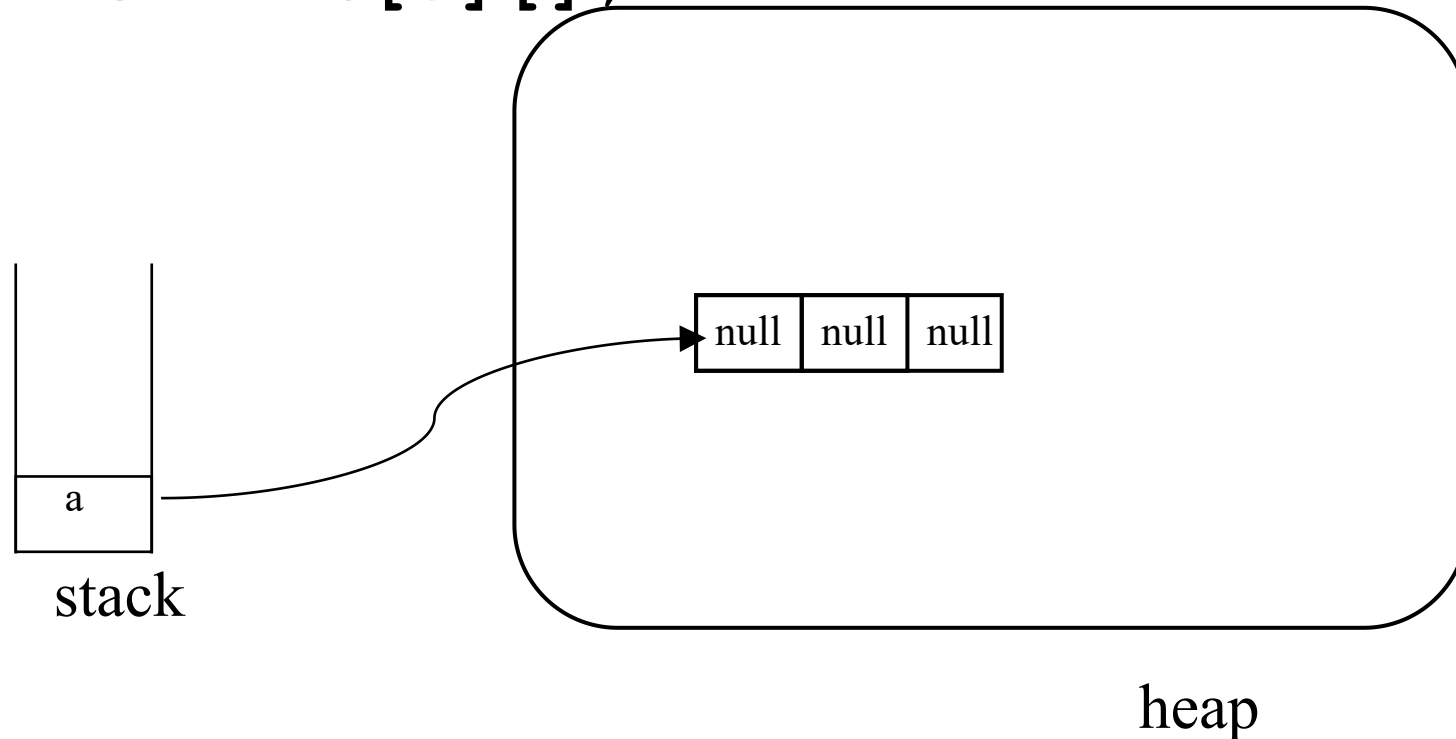
```
int[][] a = new int[2][3];
```



# Višedimenzionalni nizovi

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

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



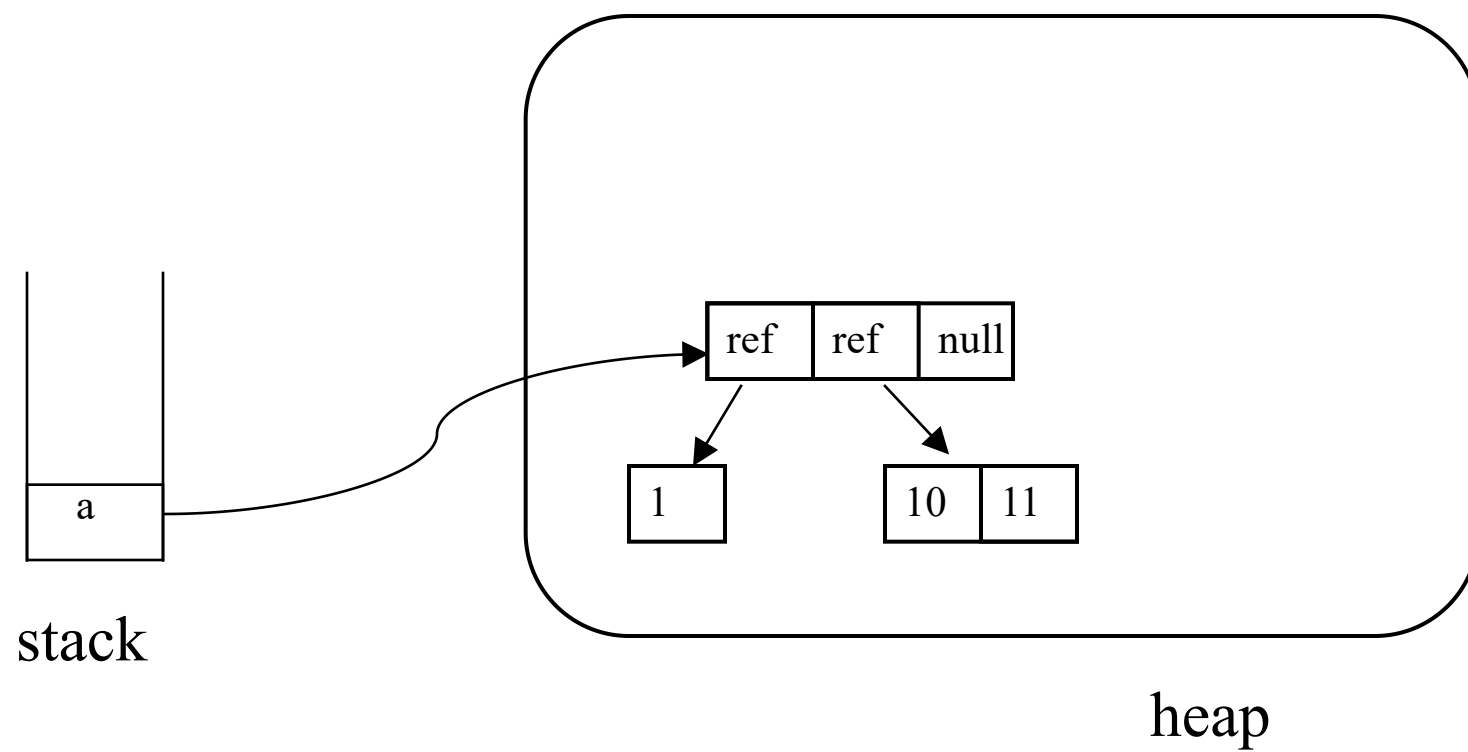
# Višedimenzijski nizovi

- Za svaki element prvog niza definiše se novi niz proizvoljne dužine
- Moguće je napraviti dvodimenzijski niz čije vrste imaju različiti broj kolona u svakoj vrsti

```
int[][] a = new int[3][];  
a[0] = new int[1];  
a[0][0]=1;  
a[1] = new int[2];  
a[1][0]=10;  
a[1][1]=11;  
//šta bi bila vrednost a[2] ?
```



# Višedimenzionalni nizovi



# Prolazak kroz elemente višedimenzijskih 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();  
}
```

# Formati ispisa na ekran

- Za ispis na ekran se koriste funkcije *print* i *println* koje očekuju tekst kao parametar

- *print* – ispiši tekst
- *println* - ispiši tekst i pređi kursorom u novi red

```
System.out.print("Poruka");
```

```
System.out.println("Poruka");
```

- Između otvorene i zatvorene zagrade dozvoljeno je izvršiti
  - konkatencijua više stringova
  - konkatenciju striga sa primitivnim tipovima
  - konkatenciju stringa sa objektima (poziva se njihova *toString* metoda)

```
int ocena = 8;
```

```
System.out.println("Dobili ste ocenu: " + ocena);
```

# Formati ispisa na ekran

- Funkcija *printf* omogućuje formatizovani ispis

```
int c = 356;
```

```
System.out.printf("celobrojni: %d\n", c);
```

```
-->celobrojni: 356
```

```
System.out.printf("celobrojni: %10d\n", c);
```

```
-->celobrojni: _____356
```

```
System.out.printf("celobrojni: %+10d\n", c);
```

```
-->celobrojni: _____+356
```

```
System.out.printf("celobrojni: %+10d\n", -c);
```

```
-->celobrojni: _____-356
```

```
System.out.printf("celobrojni: %-10d\n", c);
```

```
-->celobrojni: 356_____
```

# Formati ispisa na ekran

```
//formatizovani ispis na ekran
```

```
System.out.printf("Ispis celog broja %d \n", 10);
```

```
-->Ispis celog broja 10
```

```
System.out.printf("Ispis karaktera %c \n", 'A');
```

```
-->Ispis karaktera A
```

```
System.out.printf("Ispis karaktera %c \n", 66);
```

```
-->Ispis karaktera B
```

```
System.out.printf("Ispis razlomljenog broja %f \n", 3.14);
```

```
-->Ispis razlomljenog broja 3.140000
```

```
System.out.printf("Ispis razlomljenog broja preciznosti 2 decimale %5.2f \n",  
    3.123456789);
```

```
-->Ispis razlomljenog broja preciznosti 2 decimale _3.12
```

# Ispis slova ćirilice i latinice š,ć,đ,č

- Na **windows** platformi se slova ćirilice ili latinice (š,ć,đ,č) ne mogu inicijalno sačuvati u okviru Java fajla jer je taj fajl sačuvan pod enkodingom Cp1252 koji ne podržava ta slova.
- Rešenja
  - Ukoliko se Java fajl sačuva u UTF-8 enkodingu pisanje navedenih slova je moguće
    - Potencijalni problem predstavlja kopiranje UTF-8 fajlova, gde se pri kopiranju kopija čuva pod Cp1252 enkodingom i napisana slova ćirilice ili latinice se gube
    - Napisati problematična slova korišćenjem njihovih UNICODE brojnih oznaka
      - đ - "\u0111", Đ- "\u0110", š - "\u0161", Š- "\u0160"
      - ć - "\u0107", Ć- "\u0106", č- "\u010D", Č- "\u010C"
      - ž - "\u017E", Ž- "\u017D",
      - Više na <http://www.fileformat.info/info/unicode/char/search.htm>

# Ispis slova ćirilice i latinice š,ć,đ,č

- Za ispis Unicode karaktera u okviru konzole potrebno je:
  - Postaviti podešavanje za pokretanje java fajla
    - *Desni klik na klasu->Run As->Run Configurations...*  
->pa iz liste pokrenutih programa selektujete željeni
    - Nad željenim programom sa desne strane odaberite karticu *Common->Encoding*
    - Odaberite stavku *Other* i unesite vrednost UTF-8
  - ili podešavanje za pokretanje eclipse alata
    - na sam kraj fajla eclipse.ini ubaciti red `-Dfile.encoding=UTF-8`

# Klase StringBuffer i StringBuilder

- Izmena stringa konkatencijom ili dodelom novog string literala kreira se novi objekat na *heap* memoriji – alternativa *StringBuffer* ili *StringBuilder* klasa.
- Omogućavaju kreiranje teksta koji se može proširiti
- *StringBuffer* metode su sinhronizovane, dok metode *StringBuilder* nisu (manji overhead = efikasniji)  

```
StringBuffer buf = new StringBuffer("Pocetni tekst ");  
buf.append("dodatni tekst 1");  
buf.append("dodatni tekst 2");  
String konacniTekst = buf.toString();
```
- Koristi uvek *StringBuilder* osim ako je potrebno deliti tekst između programskih niti.



# Klasa StringTokenizer

- slične namene kao metoda split() klase String
- "cepa" osnovni string na delove po zadatom delimiteru/delimiterima
  - originalni string se ne menja
  - parametar je tekst koji se deli
  - delovi se dobijaju pozivom metode objekta tipa StringTokenizer

# Klasa StringTokenizer

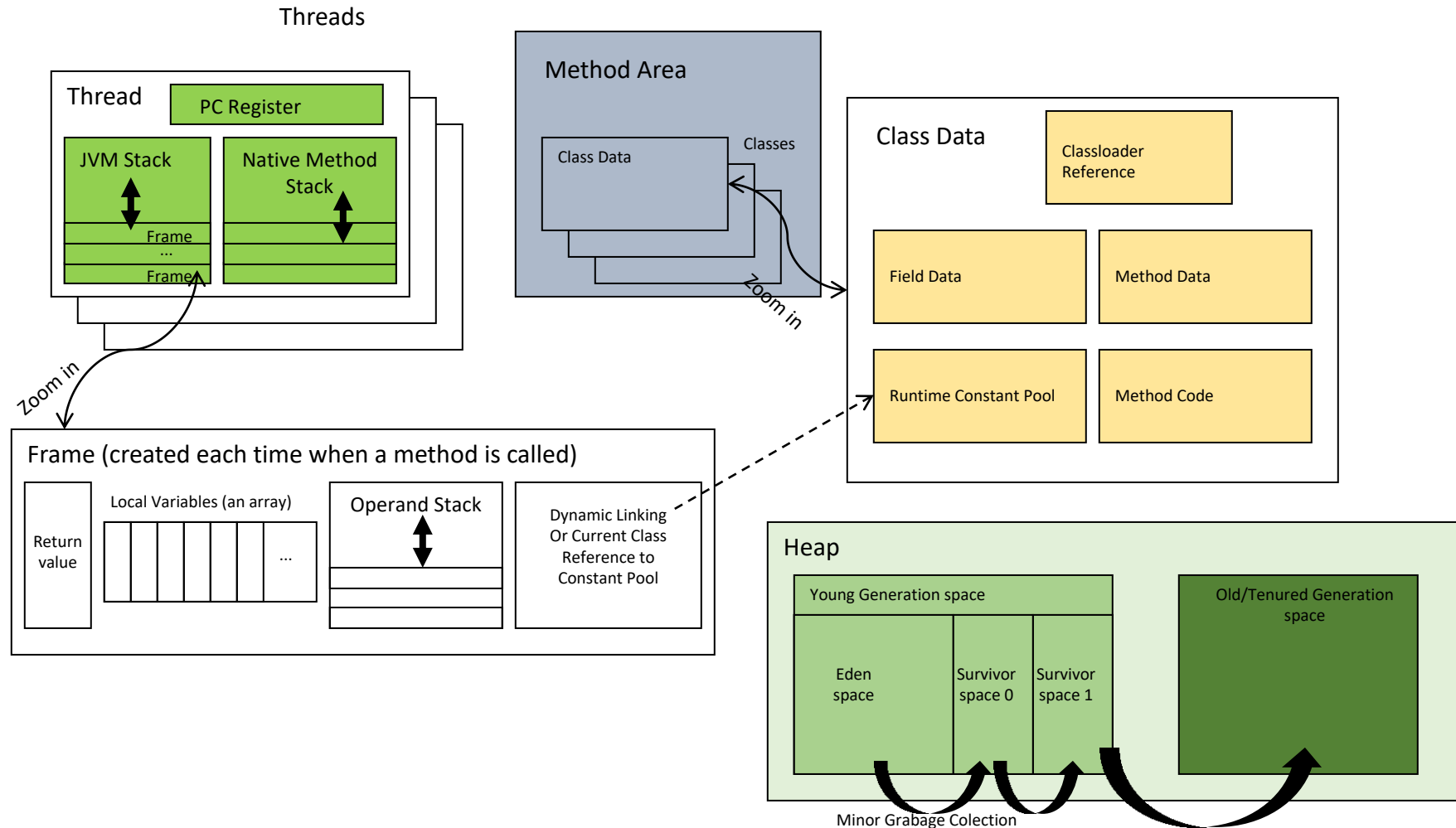
- Postoje dva načina (konstruktora) za kreiranja objekta
  - Konstruktor sa jednim parametrom i predefinisanim setom delimitera " \t\n\r\f" (razmak, tab, novi red, carriage-return, form-feed)
  - Konstruktor sa dva parametra, pri čemu je drugi parametar test u kome su navedeni delimiteri

```
StringTokenizer st = new StringTokenizer("this  
is a test", " ");  
while (st.hasMoreTokens()) {  
    System.out.println(st.nextToken());  
}
```

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

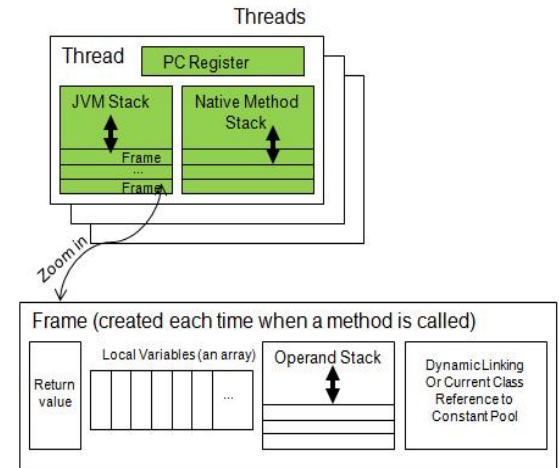
- Zvanična java dokumentacija navodi nekoliko različitih prostora za skladištenje podataka u toku rada JVM i za izvršavanja Java programa
- Prostor se deli grubo u dve kategorije
  - Prostor za podatke koji se kreira za svaku programsku nit (1 Java program može pokrenuti više programskih niti)
  - Prostor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

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



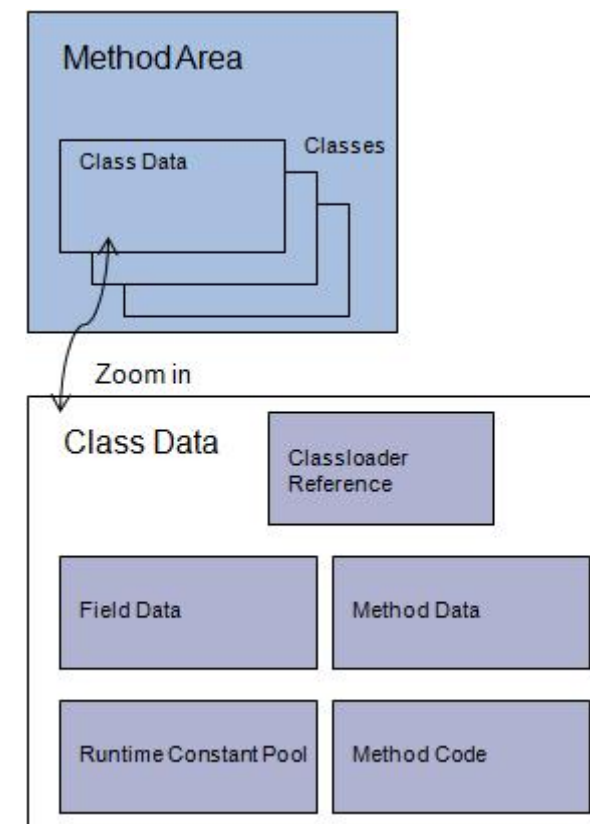
# Memorija za Programske niti

- 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



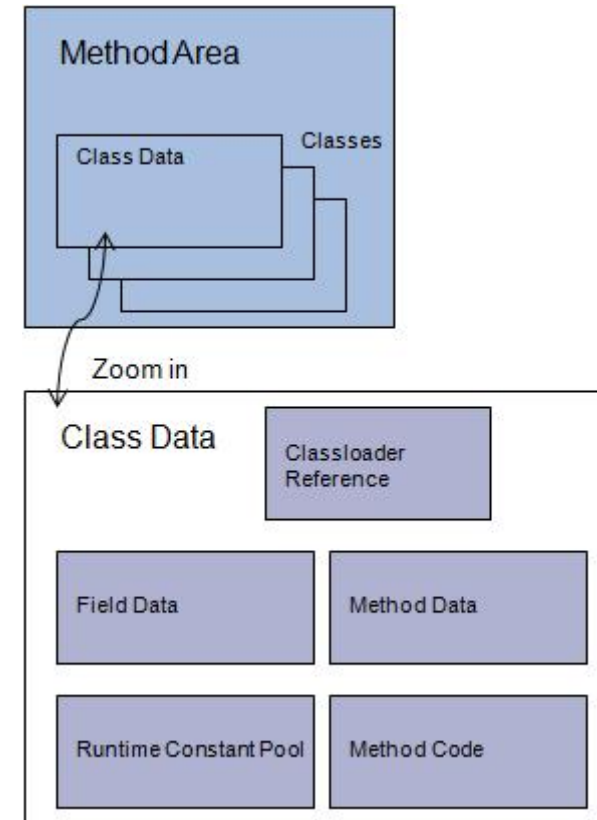
# Deljena Non-Heap memorija – Method Area

- *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

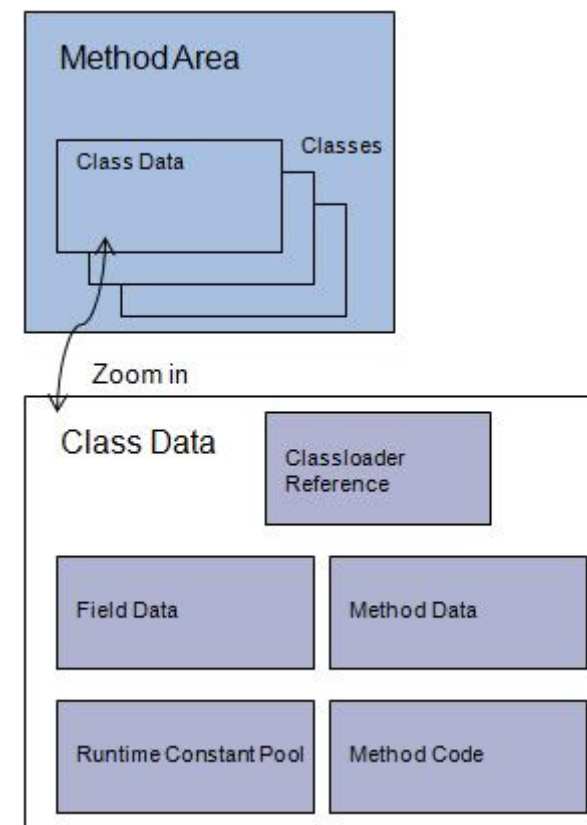
- *Field Data* za svaki atribut klase sadrži informacije za ime, tip, modifikatore pristupa i dodatne vrednosti.
- *Method Data* za svaku metodu sadrži informacije za ime, povratnu vrednost, tipove parametara, modifikatore pristupa i dodatne attribute
- *Method Code* za svaku metodu sadrži prevedeni 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

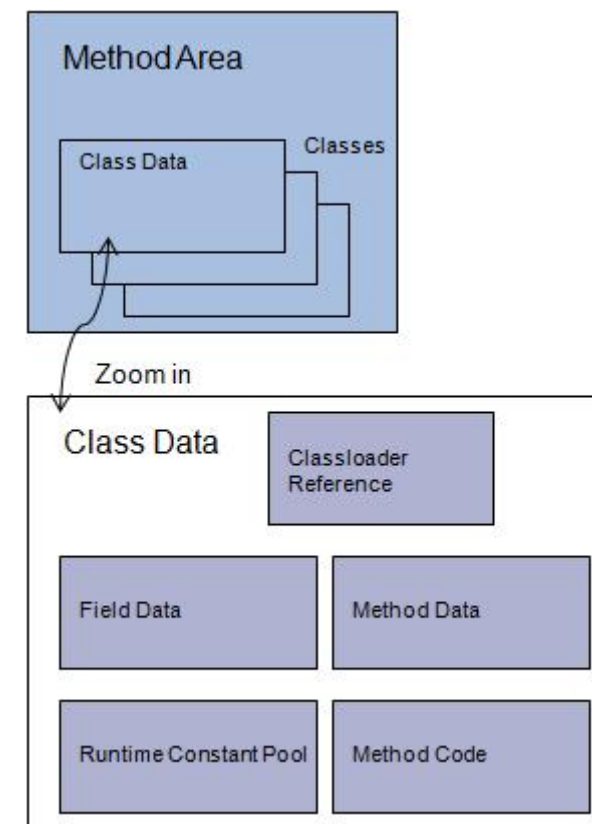
```
Last modified Dec 2, 2016; size 583 bytes
MD5 checksum aa5b26ab5a04e7e66138c06710368abd
Compiled from "MojaKlasa.java"
public class SinisinPaket.MojaKlasa
  minor version: 0
  major version: 51
  flags: ACC_PUBLIC, ACC_SUPER
Constant pool:
  #1 = Class                #2          // SinisinPaket/MojaKlasa
  #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

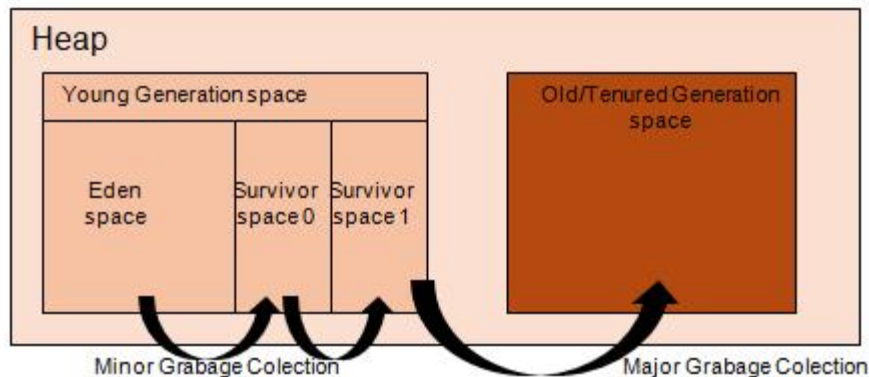


Simbolička oznaka #23 sadrži vrednost testa “String vrednost zadata direktno u kodu”



# Deljena Heap memorija

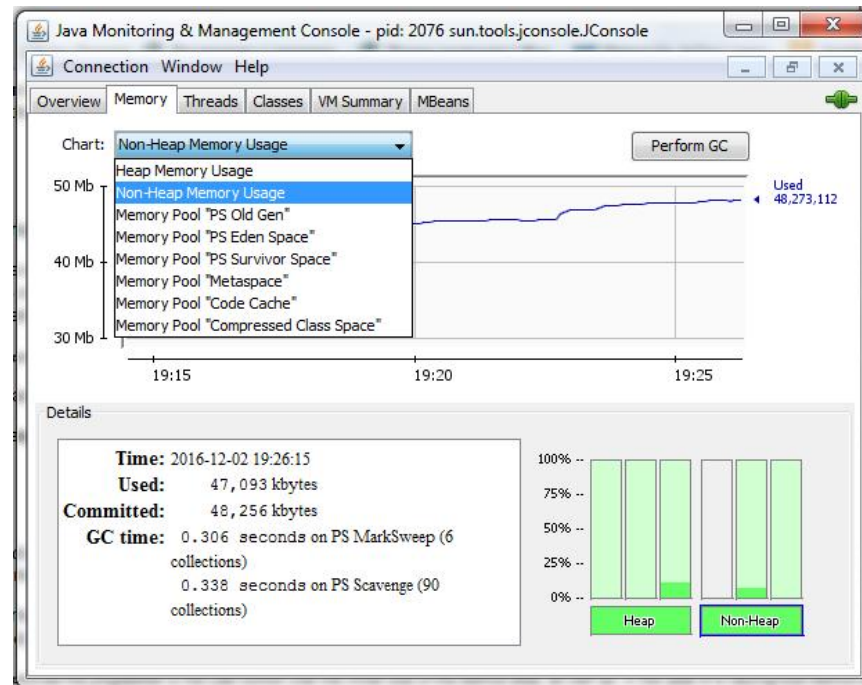
- *Heap* (dinamička memorija) deo memorije se deli između svih niti JVM
- je memorija gde se smeštaju dinamički alocirane vrednosti (vrednosti mogu da menjaju veličinu)
- smeštaju se vrednosti instance klasa i nizovi
- Vrednosti sa *heap*-a se ne brišu kada se završi metoda (kao na steku), već njih uklanja 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 Cashe* 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.

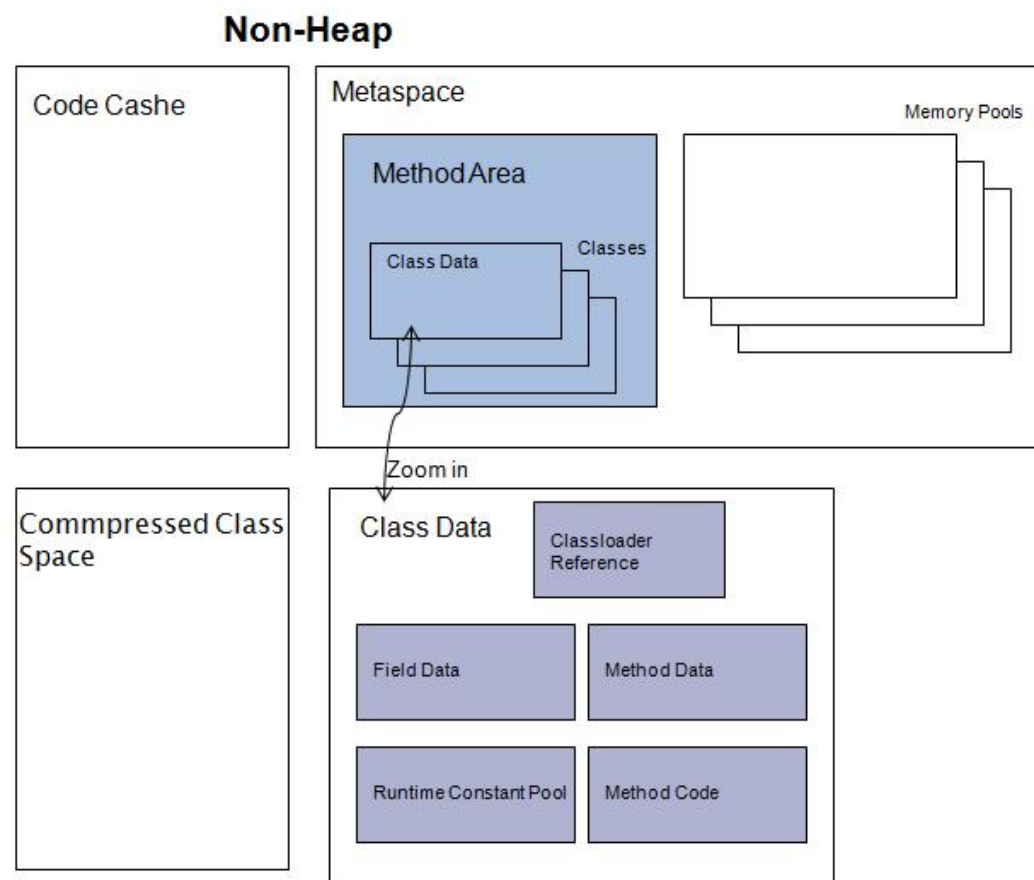


☞ *Compressed Class Space* se takođe koristi za skladištenje metapodataka o klasama.

☞ *Code Cashe* 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 Memorijske ukljuživši 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 izdvojen na delove i upravljanje memorijom na stack-u je po *LIFO (Last-In-First-Out)* principu, dok je heap memorija izdvojena 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*.

# Klasa String

- Niz karaktera je podržan klasom String. String nije samo niz karaktera – **on je klasa!**
- Od java 1.7 skladište se na heap memoriji u delu *String pool*.
- **Objekti klase String se ne mogu menjati (*immutable*)!**
- *Immutable Objects* je objekat kome se definiše vrednost u trenutku njegovog kreiranja. Za njega ne postoje metode, ni načini kako da se ta vrednost dodatno promeni.

# Klasa String

- Prethodno omogućava optimizaciju memorije. U slučaju da više string promenljivih imaju isti tekstualni sadržaj tada se za njih kreira samo jedna vrednost u *String pool*-u (optimizacija) i sve promenljive dobijaju referencu ka toj vrednosti.
- To bi značilo da će p1, p2 i p3 pokazivati na istu vrednost u *String pool*-u.

```
String p1= "Tekst je ovo";  
String p2= "Tekst " + "je ovo";  
String temp = " je ";  
String p3= "Tekst" + temp+ "ovo";
```