Akademija tehničko-vaspitačkih strukovnih studija
odsek NIŠ

Katedra za Informaciono-komunikacione tehnologije

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Prof. dr Zoran Veličković, dipl. inž. el.

2019/2020.

Prof. dr Zoran Veličković, dipl. inž. el.

OBJEKTNO ORIJENTISANO PROGRAMIRANJE - OOP

Upravljanje pristupom, ugnežđene i anonimne klase, vararg metode

(8)

Sadržaj

- **UPRAVLJANJE PRISTUPOM**
  - Upravljanje pristupom i nasleđivanje
  - Paketi/klase/metode
  - Primer steka za 10 celobrojnih članova
- **REZERVISANE REČI U JAVI**
  - `static`
  - `this`
  - `final`
  - `static`
- **UGNEŽDENE KLASE**
  - Spoljna i unutrašnja klasa
- **ANONIMNE KLASE I FUNKCIJE**
  - Anonimna klasa kao parametar
- **ARGUMENTI SA KOMANDNE LINIJE**
  - Argumenti promenljive dužine
  - `vararg` - ograničenja

Upravljanje pristupom (1)

- Već znamo da se **KAPSULIRANJEM** međusobno povezuju **PODACI** i njima pripadajući **PROGRAMSKI KOD**.
- Ovo ima za posledicu da se **KAPSULIRANJEM** može uticati na to **KOJI DELOVI** programa imaju **PRAVO PRISTUPA** članovima klase.
- **KONTROLISANJE PRISTUPA** članicama klase sprečavaju se **NENAMERNE GREŠKE** i **ZLOUPOTREBE**.
- Dakle, **KLASA** je "**CRNA KUTIJA**" koja može da se koris KAO TAKVA, ali se njenoj unutrašnjosti **NE MOŽE** neovlašćeno pristupiti.
- Način pristupa članovima klase se primarno određuje **SPECIFIKATOROM PRISTUPA** koji se zadaje pri **DEKLARACIJI** klase, metode ili promenljive.
- Takođe, **NASLEĐIVANJE KLASA** (predavanje br. 7) i pripadnost **PAKETIMA** (predavanje br. 4) imaju uticaja na **REGULISANJE** pravima pristupa članovima klase.

Upravljanje pristupom (2)

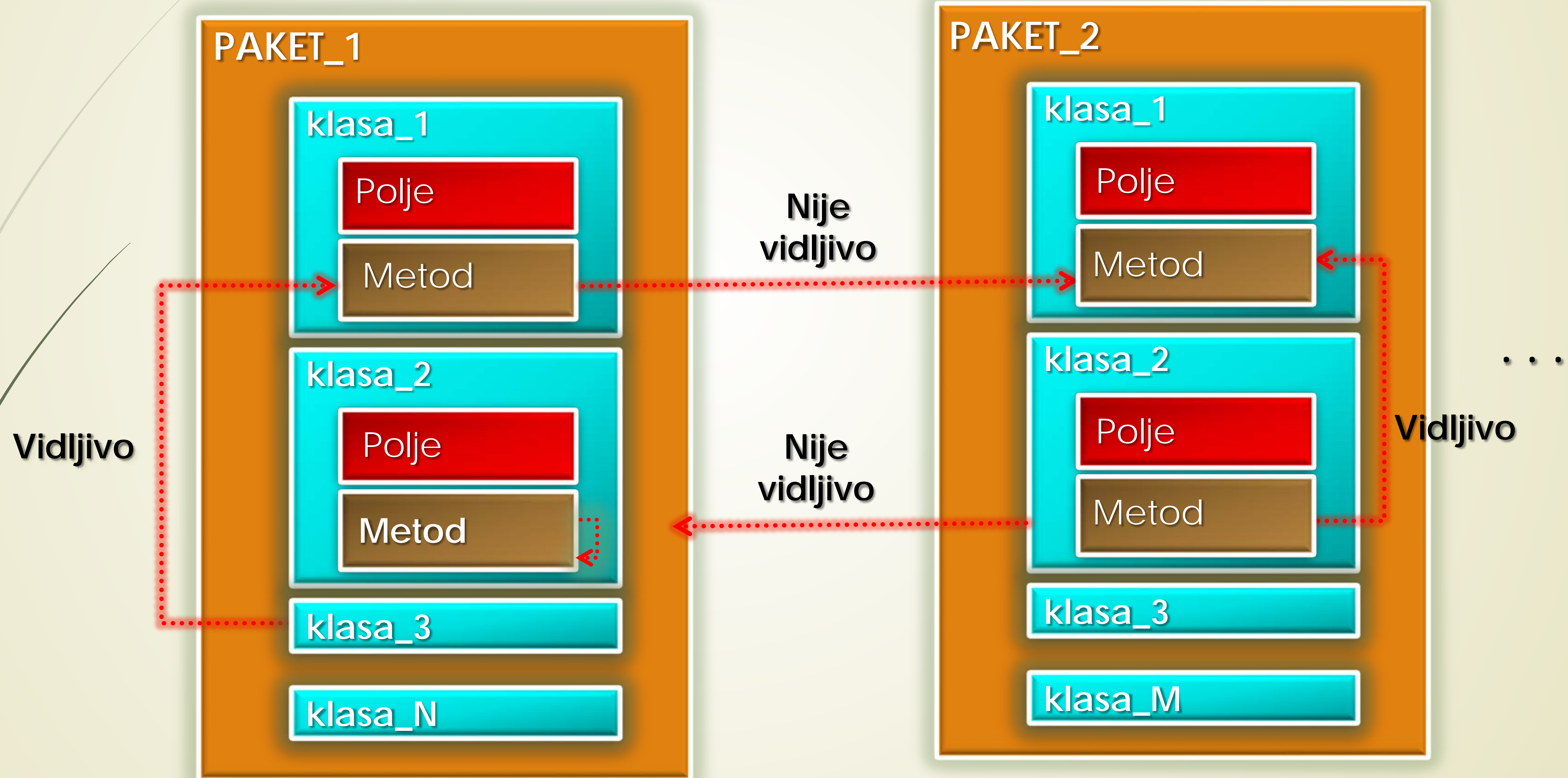
- Kod Jave su definisani sledeći specifikatori pristupa: `public`, `private` i `protected`.
- Takođe, postoji i **PODRAZUMEVANI** (nespecifirani) nivo pristupa.
- Ako se **ČLAN KLASE** označi specifikatorom pristupa private, tada njemu mogu pristupiti samo drugi članovi **ISTE KLASE**.
- Sada bi trebalo da je jasno zašto ispred metode `main()` uvek treba – mora da stoji specifikator pristupa `public`.
- Osnovna ideja **KAPSULIRANJA** je zapravo da se **OGRANIČI PRISTUP** podacima u klasi.
- Često se definišu i **METODE** koje su **PRIVATNE** i **UNUTAR KALSE**.
- **SPECIFIKATOR PRISTUPA** prethodi specifikaciji **TIPA** člana klase.
- Već znate da se upravo **SPECIFIKATOROM PRISTUPA POČINJE** specifikacija člana klase.
- Ako se **NE NAZNAČI** (izostavi) specifikator pristupa, pristup članovima klase je **JAVAN UNUTAR ISTOG PAKETA** i biće **ONEMOGUĆEN** izvan paketa.

Kontrola pristupa i nasleđivanje

SPECIFIKATOR PRISTUPA	private	bez spec.	protected	public
Ista KLASA	Da	Da	Da	Da
Potklasa istog PAKETA	Ne	Da	Da	Da
KLASA istog pak. koje nisu potklase	Ne	Da	Da	Da
POTKLASE iz drugog PAKETA	Ne	Ne	Da	Da
KLASE iz drugih PAKETA koje nisu POTKLASE	Ne	Ne	Ne	Da

Da – dozvoljava se pristup Ne – ne dozvoljava se pristup

Paketi/klase/metode



Primer: Upravljanje pristupom (1)

```
class Test {                      // Klasa Test
    int a;                        // BEZ specifikacije, dakle javna u svom paketu!
    public int b;                 // javni pristup
    private int c;                // PRIVATNI pristup, pristup moguć samo iz kl. Test

                                  // Dve metode za pristup (čitanje i postavlj.) prom. c
    void set_c(int i) {           // metoda za postavljanje vrednosti za promenljivu c
        c = i;
    }
    int get_c() {                 // vraćanje (čitanje) vrednosti promenljive c.
        return c;                 // Za pristup c-u
    }                             // koriste se metode get_c/set_c
}                                 // Kraj klase Test
```

Primer: Upravljanje pristupom (2)

```
class AccessTest {
  public static void main(String args[]) {
    Test ob = new Test();

     ob.a = 10;
     ob.b = 20;

     ob.c = 100;

     ob.set_c(100);


    System.out.println ("a, b, i c: " + ob.a + " " +  ob.b + " " + ob.get_c() );
  }
}
```

Kreiranje objekta **ob** tipa **Test** sa prethodnog slajda

Ovo je **U REDU**, promenljivama **a** i **b** se može pristupiti **DIREKTNO**

Ovo **NIJE OK** i izazvaće se **GREŠKA!**

OK. Može se pristupiti promenljivoj **c** preko njenih **PRISTUPNIH METODA**

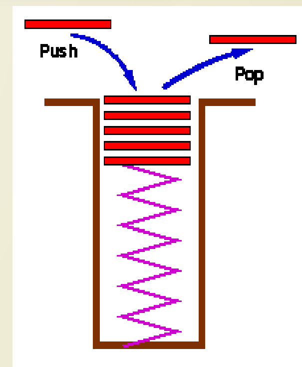
Klasa **Stek** za 10 celobrojnih članova

```
class Stack {
private int stck[] = new int[10];
private int tos;
  Stack () {
    tos = -1;
  }
  void push (int item) {
    if(tos == 9)
      System.out.println ("Stek je popunjen.");
    else
      stck[++tos] = item;
  }
  int pop() {
    if(tos < 0) {
      System.out.println ("Stek je prazan.");
      return 0;
    }
    else
      return stck[tos--];          }   }   // Kraj klase stack
```

stck (stek) i tos (vrh steka) su privatne promenljive

Inicijalizacija vrha steka, **KONSTRUKTOR**

Metoda **push** za stavljanje stavke na stek



Metoda **pop** za uzimanje stavke sa steka

Primena klase **Stek**

```
class TestStack {
public static void main (String args[]) {
    Stack mystack1 = new Stack();
    Stack mystack2 = new Stack();
    for(int i=0; i<10; i++)   mystack1.push(i);
    for(int i=10; i<20; i++)  mystack2.push(i);
    System.out.println ("Stack in mystack1:");
    for(int i=0; i<10; i++)
       System.out.println (mystack1.pop());
    System.out.println ("Stack in mystack2:");
    for(int i=0; i<10; i++)
       System.out.println (mystack2.pop());
    // mystack1.tos = -2;
    // mystack2.stck[3] = 100;
  }
}
```

Formiranje stekova, 2 objekta tipa **Stack**

Stavljanje po 10 brojeva na stekove

Uzimanje brojeva sa stekova

Ove naredbe **NISU VALIDNE**, nemoguć je pristup promenljivama, zašto?

Rezervisana reč: **`static`**

- Ako je potrebno definisati **ČLAN KLASE** koji se koristi **NEZAVISNO OD OBJEKTA** treba ga označiti kao static.
- Pažnja: U C# i Javi krljučna reč static ima **SASVIM DRUGO ZNAČENJE** nego kod C-a!
- Primer **STATIČNE METODE** je poznata metoda main(), setite se da smo uvek ovu metodu označavali **STATIČNOM**, sada je razlog tome očigledan (koristimo je bez formiranih objekata).
- Metoda **`main()`** se poziva **PRE PRAVLJENJA** bilo kakvog objekta!
- **SVE INSTANCE** neke klase dele ISTU statičku promenljivu.
- Evo nekoliko **OGRANIČENJA** statičnih metoda:
  - Statične metode mogu pozivati **SAMO** statične metode;
  - Statične metode moraju pristupati **SAMO** statičnim podacima;
  - NE MOGU se koristiti rezervisane – ključne reči this i super (ove ključne reči su obrađene na prethodnom predavanju vezanom za nasleđivanje klasa).

Rezervisana reč: **`this`** (1)

- Ponekad je potrebno da **METODA UKAŽE NA OBJEKT** od koga je i sama **POZVANA**.
- U Javi se može ukazati na **POZIVNI OBJEKT** rezervisanom rečju this.
- Takođe, rezervisana reč this se može upotrebiti u **BILO KOJOJ METODI** da se ukaže na **TEKUĆI OBJEKT**.
- Dakle, **`this`** je referenca na objekt za koji je pozvana metoda.
- Ovo će omogućiti korišćenje **FORMALNIH PARAMETARA METODA** sa ISTIM IMENOM koje poseduju promenljive instance.
- Kada lokalna promenljiva ima ISTO IME kao promenljiva instance ona SKRIVA promenljivu instance.
- Rezervisana reč this omogućava DIREKTNO OBRAĆANJE OBJEKTU i njome se razrešava PROBLEM ISTOG IMENA promenljivih.

Rezervisana reč: **this** (2)

```
class Kutija {
    double širina;
    double visina;
    double dubina;
 Kutija(double širina, double visina, double dubina) {
    this.širina = širina;
    this.visina = visina;
    this.dubina = dubina;
 }
 double zapremina () {
    return širina*visina*dubina;
 }
}
```

ISTA IMENA promeljivih instanci i lokalne promenljive

Rezervisane reči **this** su u ovom primenu **SUVIŠNE**, ali su vrlo korisne - deskriptivne

Rezervisana reč **this** razrešava **PROBLEM ISTIH IMENA**

Rezervisana reč: `final`

- **PROMENLJIVA** deklarisana ključnom rečju `final`, **NE MOŽE SE MENJATI**!
- Po nepisanom PRAVILU promenljive označene kao final, pišu se **VELIKIM SLOVIMA**:

```
final int NAPRAVI = 10;
```

- Ovakve promenljive **NE ZAUZIMAJU MEMORIJU** u svakoj **INSTANCI** klase.
- Zapravo, ova vrsta promenljivih se može smatrati **KONSTANTOM**.
- Rezervisana reč final se može primeniti i na **METODE**, ali tada ima drugo značenje.
- Upotrebu rezervisane reči final će imati značajnu ulogu prilikom **NASLEĐIVANJA KLASA** (nasleđivanje klasa je obrađeno na prethodnom predavanju).
- Setite se, ovom rečju se **ONEMOGUĆAVANJE** nasleđivanje te klase.

Static promenljive, metode, ...

```
class UseStatic {
  static int a = 3;
  static int b;
  static void meth (int x) {
    System.out.println("x = " + x);
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
  static {
    System.out.println("Statički blok je inicijalizovan.");
    b = a * 4;
  }
  public static void main (String args[])
  {
    meth(42);
  }
}
```

STATIČKE **PROMENLJIVE**, koriste se **NEZAVISNO OD OBJEKTA**, deklarisane u klasi!

STATIČKA **METODA**, **meth()** bez referenciranja na određenu instancu

STATIČKI BLOK za inicijalizaciju {...}.

Koriste se statičke promenljive

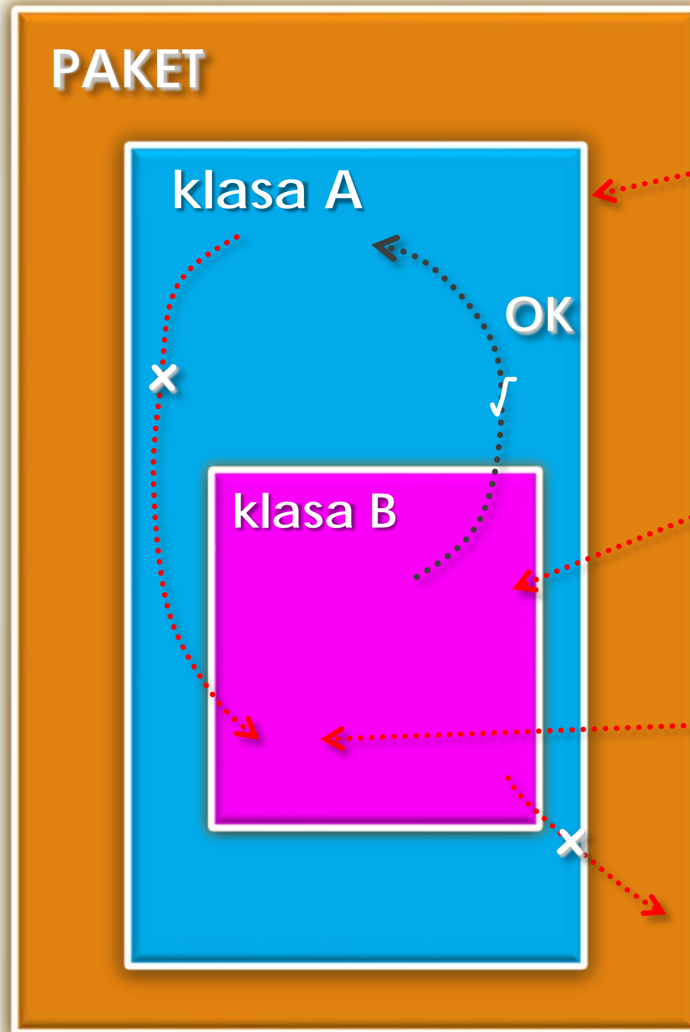
Korišćenje metode **BEZ KREIRANJA** odgovarajućeg objekta!

Izlaz:
x=42
a=3
b=12

Ugnežđene i unutrašnje klase (1)

- U nekim slučajevima, dobro je definisati klasu **UNUTAR KLASE**, tada se ona naziva **UGNEŽĐENOM KLASOM**.
- **OBLAST VAŽENJA** ugnežđenih klasa ograničena je oblašću važenja **OBUHVATAJUĆE KLASE**.
- Ako je klasa B definisana unutar klase A, tada klasa A zna za klasu B, ali za nju se **NE ZNA IZVAN KLASE A**.
- UGNEŽĐENA KLASA IMA PRISTUP članovima klase u koju je ugnežđena, uključujući i **PRIVATNE ČLANOVE**!
- **OBUHVATAJUĆA KLASA NEMA PRISTUP** članovima ugnežđene klase.
- Postoje **DVA TIPA** ugnežđenih klasa:
  - STATIČNE, sadrži modifikator **`static`** - retko se koriste.
  - NESTATIČNE (unutrašnje), IMAJU PRISTUP SVIM PROMENLJIVAMA I METODAMA svoje spoljašnje klase.

Ugnežđene i unutrašnje klase (2)



- Ako je klasa B definisana unutar klase A, tada klasa A **ZNA** za klasu B, ali za nju se **NE ZNA** izvan klase A.
- **UGNEŽĐENA** klasa **IMA PRISTUP** članovima klase u koju je ugnežđena, uključujući i privatne članove.
- **OBUHVATAJUĆA** klasa **NEMA PRISTUP** članovima ugnežđene klase.

Primer: unutrašnje/spoljne klase (1)

```
class Spoljna {
   int spoljna_x = 100;

   void test () {
         Unutrasnja unutrasnja = new Unutrasnja();
         unutrasnja.prikazi();
   }

   // ovo je unutrašnja klasa
         class Unutrasnja  {
              void prikazi() {
               System.out.println(“prikaz: spoljna_x = " + spoljna_x);
             }
         }
}
```

Promenljiva spoljne klase

SPOLJAŠNA KLASA
obuhvata unutrašnju klasu

Pravi se instanca klase **Unutrasnja** i poziva njenu metodu **prikazi()**

UNUTRAŠNJA KLASA
definisana unutar spoljašnje

Iz klase **Unutrasnja** **MOŽE** se pristupiti promenljivoj **spoljna_x**

Primer: unutrašnje/spoljne klase (2)

```
class InnerClassDemo {
  public static void main (String args[]) {
    Spoljna spoljna = new Spoljna();
    spoljna.test();
  }
}
```

Pravi se instanca klase **Spoljna** i poziva sopstvenu klasu **test().**

Metoda **test()** pravi instancu klase **Unutrasnja** i poziva svoju metodu **prikazi()**.

Klasa **Unutrasnja** poznata samo unutar oblasti važenja klase **Spoljna!**

Izlaz:
prikaz: spoljna_x = 100

Primer: Nedozvoljen poziv (1)

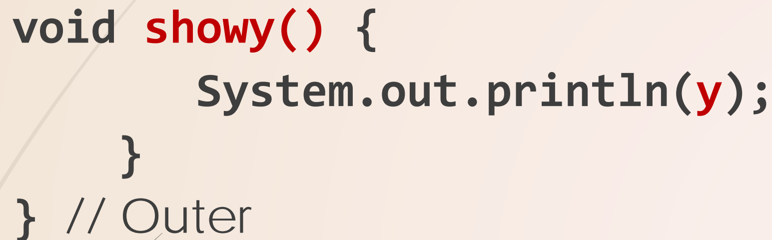
```
class Outer {
  int outer_x = 100;
  void test() {
    Inner inner = new Inner();
    inner.display();
  }

     class Inner {
      int y = 10;
        void display() {
           System.out.println ("display: outer_x = " + outer_x);
        }
     }
```

Spoljašna klasa, **Outer**

Unutrašnja klasa, **Inner**

y je promenliva instance - lokalna promenljiva za **Inner** klasu i **NE MOŽE** se koristiti izvan ove klase!

Primer: Nedozvoljen poziv (2)



```
void showy() {
        System.out.println(y);
    }
} // Outer
```

Definisano u spoljašnjoj klasi

GREŠKA!
y nije poznato ovde!

```
Testiranje:
.................................................

 class InnerClassDemo {
   public static void main (String args[]) {
     Outer outer = new Outer();
     outer.test();
   }
}
```

- **UGNEŽDENE KLASE** su posebno značajne prilikom **OBRADE DOGAĐAJA** u **APLETU** (detaljnije o apletima pred kraj kursa).

Anonimne klase i funkcije (1)

- **ANONIMNE KLASE** omogućavaju da se izvorni Java kod napiše **SAŽETIJE** i **EFIKASNIJE**.
- **ANONIMNE KLASE** se koriste ako je **SAMO JEDNOM** (na tom mestu) potrebno koristiti **KLASU** ili **FUNKCIJU**.
- Anonimne klase se **ISTOVREMENO** i **DEKLARIŠU** i **INSTANCIRAJU**.
- **ANONIMNE FUNKCIJE** i **KLASE** se upotrebljavaju za definisanje funkcije ili klase neposredno **PRE KORIŠĆENJA** - upravo na mestu u izvornom kodu gde se i **KORISTE** (to ih onemogućava da budu ponovno korišćene).
- **ANONIMNE FUNKCIJE** i klase **NISU** deklarisane izvan tela pozivne funkcije i postavljaju se u izvornom kodu BEZ IMENA (otuda i potiče naziv).
- Da bi se koristila anonimna klasa, mora se **NASLEDITI** (proširiti) **BAZNA KLASA** ili se mora **IMPLEMENTIRATI INTERFEJS** (o interfejsima na sled. predavanju).
- **ANONIMNA KLASA** je **NEIMENOVANA - IZVEDENA KLASA** koja implementira ili proširuje funkcionalnost bazne klase.

Anonimne klase i funkcije (2)

```
public class MainClass {
  static class OutputL {
       public void output() {
            System.out.println("Nije podržano ...");
         }
  }

public static void main (String[] args) {
   OutputL regularnaInstanca = new OutputL();
      OutputL anonymousClass = new OutputL() {
        public void output() {
            System.out.println("Alternativa ne postoji.");
       }
   };
   regularnaInstanca.output();
   anonymousClass.output();
  }
}
```

Dekleracija **BAZNE** klase **OutputL**

Metod **output** je jedini metod klase **OutputL**

Objekt klase **OutputL**

anonymousClass je anonimni objekt klase **OutputL** sa svojom metodom **output()**

Dekleracija anonimne unutrašnje klase se smatra iskazom i sintaksa **(;)** dolazi posle zatvaranja kodnog bloka.

Poziv **output()** metoda za regularni i anonimni objekt

Anonimna klase kao parametar (1)

- Primer: Ako se želi izvršenje neke operacije nad dva cela (int) broja i vraćanje dobijenog rezultata, može se efikasno koristiti **ANONIMNA KLASA**.

Deklaracija bazne klase **MathOperation**

Statična metoda **operation** sa dva **int** parametra

Statična metoda **performOperation** sa dva int parametra i instancom op tipa **MathOperation**

```
public class MainClass {
    static class MathOperation {
        public int operation (int a, int b) {
         return 0;
    }
}
// Metoda koja uzima objekt tipa MathOperation kao parametar.
 static int performOperation (int a, int b, MathOperation op)
    {
      return op.operation (a, b);
    }
```

Anonimna klase kao parametar (2)

```
public static void main(String[] args) {
    int x = 100; int y = 97;
    int resultOfAddition = performOperation (x, y, new MathOperation() {
      public int operation (int a, int b) {
        return a + b;
      }
  });
    int resultOfSubtraction = performOperation (x, y, new MathOperation() {
      public int operation(int a, int b) {
        return a - b;
      }
  });
    System.out.println("Addition: " + resultOfAddition);
    System.out.println("Subtraction: " + resultOfSubtraction);
  }
}
```

Dve int promenljive:
resultOfSubtraction i **resultOfAddition**

Preklapanje metoda
opeartion()

Argumenti sa komandne linije

- U C-u je već poznato da se parametri main() metodi mogu proslediti sa **KOMANDNE LINIJE**.
- Argumenti sa komandne linije se prosleđuju programu **U TRENUTKU POKRETANJA**.
- U Javi (a i C#) se slanje argumenata metodi main() sa komandne linije takođe može obaviti na **SLIČAN NAČIN**.
- Argumenti koji se prosleđuju metode main() se upisuju na **KOMANDNOJ LINIJI** odmah **IZA IMENA PROGRAMA**.
- Argumenti se smeštaju kao **ZNAKOVNI NIZ** koji se kasnije prosleđuje metodi **`main()`**.
- Primer:

Niz stringova

```
Java Ime_klase argumenti_komandne_linije
```

Primer: argumenti sa komandne linije

```
// Primer prikaza svih argumenata sa komandne linije.
class CommandLine {
  public static void main (String args[]) {
    for(int i=0; i < args.length; i++)
      System.out.println ("args[" + i + "]: " +  args[i]);
  }
}
```

Štampaj argumente iz niza **args**

Isčitati sve argumente sa komandne linije, ima ih **args.length**.
Koristi se svojstvo **length** referentne string promenljive

Argumenti promenljive dužine - `vararg`

- **J2SE5** obezbeđuje **NOVI MEHANIZAM** metodama koje treba da prihvate **PROMENLJIV BROJ** argumenata.
- Metode koje koriste argument promenljive dužine varag (engl. variable length argument) se nazivaju **VARARG METODE**.
- Nova metoda **`printf()`** koja je deo Javine U/I biblioteke upravo koristi pogodnosti **`vararg`** metoda.
- Argument promenljive dužine se zadaje operatorom "**tri tačke**" **`(...)`.**
- Na sledećem slajdu je prikazana upotreba ovog operatora u **`vaTest()`** metodi.
- Primer dekleracije metode sa vararg parametrom:

```
static void vaTest(int ... v)
```

Primer: Argumenti promenljive dužine

```
class Arg_prm_duz {
static void vaTest(int ... v) {
    System.out.print("Br. arg.: " + v.length + "Sadržaj:")
        for(int x: v)
         System.out.print(x + " ");
         System.out.println();
}
public static void main (Sting args[])
  {
    vaTest(10);
    vaTest(1,2,3);
    vaTest();
  }
}
```

v se smatra nizom

Foreach petlja, uzimaju se vrednosti iz niza **v**

Poziv **vaTest()** metode sa **RAZLIČITIM** brojem argumenata

Izlaz:
Br. arg.: 1 Sadržaj 10
Br. arg.: 3 Sadržaj 1 2 3
Br. arg.: 0 Sadržaj

vararg - ograničenja

- **VARARG METODE** pored varag parametra mogu imati i “**normalne**” parametre.
- Međutim, vararg parametar **MORA BITI POSLEDNJI** u nizu koje metoda deklariše.

```
int uradi(int a, int b, double c, int ... vrednosti)
```

- Argument promenljive dužine **MORA BITI POSLEDNJI** u dekleraciji.

```
int uradi(int a, int b, double c, int ... vrednosti, boolen d)
```

- Metoda može imati **SAMO JEDAN** vararg argument.
- **DOZVOLJENO JE PREKLAPANJE** vararg metoda, naravno pod uobičajenim uslovima preklapanja (u smislu **BROJA** i TIPA argumenata).

```
static void vaTest(int ... v)
static void vaTest(boolean ... v)
```