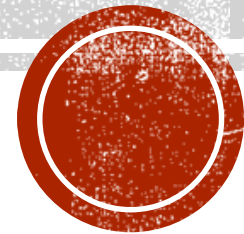


ОБЈЕКТНО ОРИЈЕНТИСАНО ПРОГРАМИРАЊЕ ПРОГРАМСКИ ЈЕЗИК ЈАВА – 1

Класе, пакети, поља, методи и објекти у Јави



КЛАСЕ У ЈАВИ

- Креирање апстрактних типова података је фундаментални концепт објектно оријентисаног програмирања.
- Апстрактни типови података функционишу веома слично као уграђени типови.
- Како у Јави класа описује структуру објекта (податке и понашање).
- На овај начин, програмер може дефинисати нови тип који се боље уклапа у проблем који се решава и није принуђен да се ограничи само на већ предефинисане типове.
- Тиме се проширује програмски језик тако што се додају нови типови података.
- Програмско окружење за Јаву омогућује да рад са новим класама/типovima буде једнако удобан и једнако сигуран као што је то случај са предефинисаним класама/типovima.

КАРАКТЕРИСТИКЕ ТИПОВА/КЛАСА У ЈАВИ

- строго типизиран програмски језик;
- објектно оријентисан програмски језик са хијерархијском структуром класа;
- не допушта декларације ван тела објекта;
- програмски код је организован по класама и сваки објекат мора бити инстанца неке класе;
- дизајнирана тако да подржава само једноструко наслеђивање;
- због ефикасности, нису баш сви елементи реализовани као објекти.

ДЕФИНИЦИЈА КЛАСЕ

- Једној Јава класи одговара једна датотека са екстензијом `.java`.
- Притом, назив класе и назив датотеке треба да буду исти.

```
<дефиниција класе> ::= {<модификатор класе>}class<идентификатор><придруживање>  
                        <тело класе>  
<модификатор класе> ::= public|abstract|final|static  
<придруживање> ::= [extends <име>][implements <листа имена>]  
<име> ::= <идентификатор>{.<идентификатор>}  
<листа имена> ::= <име>{,<име>}  
  
<тело класе> ::= {{(<дефиниција метода>|<дефиниција поља>)}}
```

- Пример класа `Zaposleni` записана у `Zaposleni.java`.

```
class Zaposleni {  
    String imePrezime;  
    double plata;  
}
```

ОБЈЕКАТ И РЕФЕРЕНЦА НА ОБЈЕКАТ

- Иако су у Јави скоро све објекти, суштински се не ради над објектима већ над референцама на објекте.
- У коду који следи се креирају две референце на инстанцу класе `Zaposleni` (променљиве `z1` и `z2`) и три референце на ниску (променљиве `s1`, `s2` и `s3`).

```
class KreiranjeReferenci {  
  
    public static void main(String[] args) {  
        Zaposleni z1;  
        Zaposleni z2;  
  
        String s1;  
        String s2;  
  
        String s3;  
    }  
}
```

КРЕИРАЊЕ ОБЈЕКТА (ИНСТАНЦЕ КЛАСЕ)

- Кад год се креира референца, потребно је ту референцу повезати са конкретним објектом.
- Креирање објекта се релизује помоћу оператора **new**:
 - издваја се из хип меморије регион у који ће бити смештене вредности атрибута тј. поља тог објекта;
 - информација о адреси тог региона се враћа као резултат примене оператора **new**.
- Доста сличности са функцијом **malloc** у програмском језику **C**.
 - У оба случаја је реч о динамичкој алокацији меморије.
 - С тим што **malloc** дозвољава експлицитно читање повратне вредности, тј. адресе.
 - Програмер у Јави ту нумеричку вредност не може читати, нити њоме манипулисати.

ПРИМЕР 2

- Нека нам је на располагању већ развијена класа **Zaposleni**.
- Креирати две инстанце класе **Zaposleni** и креирати три ниске са неким вредностима.

```
class KreiranjeObjekata{  
  
    public static void main(String[] args) {  
        Zaposleni z1 = new Zaposleni();  
        Zaposleni z2 = new Zaposleni();  
  
        String s1 = new String();  
        String s2 = new String("Браћа Бамбалић");  
        String s3 = s2;  
    }  
}
```

ОБЈЕКТИ КЛАСЕ `Object`

- С обзиром да је у Јави подржано једноструко наслеђивање, структура која се добија применом наслеђивања је дрво.
- Корен овог дрвета је класа под називом `Object` па су све класе директни или индиректни потомци класе `Object`.
- Ова класа садржи многе значајне методе попут:
 - `toString()`
 - `equals()`
 - `hashCode()`

ПРИМЕР 3

- Написати Јава програм који креира објекат класе **Object** и потом га исписује на стандардном излазу.

```
public class TestirajObject {  
  
    public static void main(String[] args) {  
        // декларација променљиве без креирања објекта  
        Object o1;  
        // креирање објекта и подешавање да инстанца променљива o1  
        // показује на тај новокреирани објекат  
        o1 = new Object();  
        // испис објекта у текстуалном облику  
        System.out.println(o1);  
        // ово изнад је еквивалентно следећем  
        System.out.println(o1.toString());  
    }  
}
```

```
java.lang.Object@2f92e0f4  
java.lang.Object@2f92e0f4
```

ПОРЕЂЕЊЕ РЕФЕРЕНЦИ НА ОБЈЕКАТ

- Поређење две променљиве оператором `==` је увек базирано на истом принципу
 - А то је регистарско поређење садржаја тих променљивих.
- У зависности од типа променљивих, зависиће и интерпретација резултата.
 - У случају да су променљиве примитивног типа, резултат поређења ће заиста указивати на једнакост садржаја тих променљивих.
 - Ако поредимо две инстанчне променљиве, једнакост ће важити само ако указују на исту инстанцу.
- На пример, можемо имати две кутије потпуно истих димензија које, приликом поређења са `==`, неће бити једнаке.
- Да би се постигао ефекат поређења на једнакост у семантичком смислу потребно је превазићи и користити метод `equals()`.

ПРИМЕР 5

- Написати Јава програм који тестира оператор == над примитивним целобројним и над инстанциним променљивама класе **Kutija**.
- Кутија се описује са три атрибута: висина, ширина и дубина.

ПРИМЕР 5 (2)

```
public class UporediPromenljive {  
    public static void main(String[] args) {  
        int x = 123, y = 123;  
        System.out.println(x == y);  
  
        Kutija kutija1 = new Kutija();  
        kutija1.dubina = 10;  
        kutija1.sirina = 2;  
        kutija1.visina = 4;  
        Kutija kutija2 = new Kutija();  
        kutija2.dubina = 10;  
        kutija2.sirina = 2;  
        kutija2.visina = 4;  
        System.out.println(kutija1 == kutija2);  
    }  
}  
true  
false
```

ИСПИТИВАЊЕ ПРИПАДНОСТИ КЛАСИ (`instanceof`)

- Применом наслеђивања остварује се релација између поткласе и наткласе.
- Будући да је релација транзитивна, важи да ако је **A** поткласа **B** и **B** је поткласа **C** онда је и **A** поткласа од класе **C**.
- У Јави постоји оператор `instanceof` који испитује постојање поменуте релације.
 - При чему се испитивање спроводи над објектом, а не над називом класе.
- За објекат класе **A** се може рећи да припада класи **A**, али и свим њеним надкласама.
 - Такав објекат има све што имају и објекти његових надкласа (када је реч о пољима и методима), и поред тога, још неке евентуалне додатне елементе.
 - Последица је да су све класе у Јави директне или индиректне поткласе класе **Object** па и испитивање да ли је неки објекат поткласа класе **Object** увек враћа `true`.

ПРИМЕР 6

- Написати Јава програм који тестира оператор `instanceof` над објектима класе `Object` и класе `Kutija` у свим могућим комбинацијама.

ПРИМЕР 6 (2)

```
public class IspitajPripadnostKlasi {  
    public static void main(String[] args) {  
        Object o=new Object();  
        Kutija k = new Kutija();  
        k.dubina=10;  
        k.visina=2;  
        k.sirina=11;  
  
        System.out.println("Објекат класе Object припада класи Object");  
        System.out.println(o instanceof Object);  
        System.out.println("Објекат класе Object припада класи Kutija");  
        System.out.println(o instanceof Kutija);  
        System.out.println("Објекат класе Kutija припада класи Object");  
        System.out.println(k instanceof Object);  
        System.out.println("Објекат класе Kutija припада класи Kutija");  
        System.out.println(k instanceof Kutija);  
    }  
}
```

Објекат класе Object припада класи Object
true

Објекат класе Object припада класи Kutija
false

Објекат класе Kutija припада класи Object
true

Објекат класе Kutija припада класи Kutija
true

КОНВЕРЗИЈА ИЗМЕЂУ ОБЈЕКТА

- Објекат неке класе је могуће безбедно конвертовати (кастовати) у објекат било које његове надкласе, укључујући и корену класу **Object**.
- На пример, ако бисмо из класе **Kutija** имали и изведену класу **ObojenaKutija**, било би могуће написати следећи код:

```
Kutija k;  
Object o;  
ObojenaKutija ok = new ObojenaKutija();  
k = ok; // имплицитна конверзија  
o = new ObojenaKutija(); // имплицитна конверзија
```

- У питању је имплицитна конверзија, јер је свака **ObojenaKutija** јесте **Kutija** или још општије гледано **Object**.

КОНВЕРЗИЈА ИЗМЕЂУ ОБЈЕКТА (2)

- Ако се врши конверзија у супротном смеру, од општије ка специфичнијој класи, онда је у питању тзв. експлицитна конверзија:

```
Object o = new ObojenaKutija();  
ObojenaKutija ok = (ObojenaKutija) o; // ово је у реду  
Object o2 = new Kutija();  
ObojenaKutija ok2 = (ObojenaKutija) o2; // грешка приликом извршавања  
Object o3 = new ObojenaKutija();  
Kutija k = (Kutija) o3; // ово је у реду
```

- Приликом конверзије у општији тип **Object** ништа се не мења у оквиру објекта. Дакле, меморија на хипу остаје нетакнута.
- Други део примера демонстрира ситуацију у којој се врши некоректна конверзија.
 - **Kutija** се конвертује у **Object**, а потом се покушава враћање у тип **ObojenaKutija**.
- Трећи део је у реду, општији тип **Object** конвертује се назад у мање општи **Kutija**, који је и даље општији од оригиналног типа **ObojenaKutija**.

ПАКЕТИ

- Програмери групишу сличне тј. повезане типове у пакете и на тај начин избегавају конфликте у именима и контролишу приступ.
- Пакет је група повезаних типова (класа, интерфејса, енумерисаних типова итд.) за коју се обезбеђује заштита при приступу и управљање простором имена.
- На пример, класа **Object** припада пакету `java.lang` из Јава **API**, класа **Scanner** припада пакету `java.util` итд.
- Класе у Јави су организоване по пакетима. Најчешће коришћени пакети су: `java.lang`, `java.util`, `java.io`, `java.net`, `java.awt`, `javax.swing` итд.
- Примери из овог уџбеника и решења датих задатака су организована по пакетима.

ПАКЕТИ (2)

- Разлози за паковање класа и интерфејса у пакете су:
 - лакше одређивање да ли су типови повезани;
 - лакше се могу пронаћи тражени типови;
 - нема именских конфликта са другим типовима истог назива;
 - допуштање да типови унутар пакета имају неограничен приступ један другом.
- Да би могле да се креирају сопствене Јава класе у пакетима, мора се знати где се налази бајт-код преведених класа — променљива **CLASSPATH**.
- Комбиновањем путање дате помоћу **CLASSPATH** и назива пакета, Јава виртуелна машина проналази бајт-код класа са којима се оперише.
- Ако **CLASSPATH** није дефинисан, подразумева се да се класе налазе у поддиректоријуму **lib** директоријума **java** унутар конкретне инсталације Јаве.

НАРЕДБА `package`

- Процес креирања сопствених пакета се може описати у три корака.
 1. Избор имена пакета.
На пример, ако је назив домена: `math.rs`, назив пакета би требало да почне са `rs.math`.
 2. Креирање структуре директоријума (фасцикли, фолдера).
 - Ако је назив пакета из једног дела назив директоријума поклапа се са називом пакета.
 - Ако то није случај, нпр. `rs.math.oop`, главни директоријум треба да се зове `rs`, његов поддиректоријум `math` и у њему треба да постоји директоријум `oop`.
У сваки од њих се могу убацити датотеке односно класе, интерфејси итд.
 3. Постављање `package` наредбе. Ово треба да буде прва наредба Јава програма.
Нпр. за `rs.math.oop` на почетку сваке датотеке у том пакету пише `rs.math.oop;`

НАРЕДБА `import`

- Наредба `import` омогућава увоз целих пакета или појединачних датотека.
- Увоз подразумева да код свега што се увози постане доступан за коришћење унутар класе/интерфејса у оквиру којег је позван увоз.
- Следећим кодом је демонстриран увоз свега што је доступно у оквиру пакета `java.util`.

```
import java.util.*;
```

- Увоз целих пакета се не препоручује, јер оптерећује процес компилације.
- Да бисмо увезли генератор псеудослучајних бројева, може се написати:

```
import java.util.Random;
```

```
...
```

```
Random rg = new Random();
```

ПРИМЕР 7

- Написати Јава програм који генерише 10 псеудослучајних целих бројева.
- Користити наредбу `import` за увоз класе `Random`.

```
package rs.math.oop.g08.p07.uvozKlase;

import java.util.Random;

public class IspisiPseudoslucajneCele {

    public static void main(String[] args) {
        Random rg = new Random(12345);
        for(int i=0; i<10; i++)
            System.out.println(rg.nextInt());
    }
}
```

1553932502
-2090749135
-287790814
-355989640
-716867186
161804169
1402202751
535445604
1011567003

ПОЉА (АТРИБУТИ)

- Свака инстанца дате класе садржи сопствени примерак (сопствену меморијску локацију) за свако од поља које се налази у оквиру објекта.
 - Зато промена вредности поља нема утицаја на исто поље другог примерка.
- Декларација поља се састоји од три компоненте:
 1. модификатора (који се опционо појављују),
 2. типа поља (тип)
 3. и имена поља (идентификатор).

```
class PripadajucaKlasa{  
    private int polje1; // модификатор тип име  
    public String polje2; // тип може бити примитивни или објектни  
}
```

ПРИСТУП ПОЉИМА

- Вредности датог поља у оквиру инстанце се може приступити само ако се референцира инстанца која садржи ту променљиву.
- Пољима датог објекта приступа се преко тзв. тачка-нотације.

`referencaNaObjekat.polje`

ПРИМЕР 8

- Дефинисати класу `Ucenik` са подацима о имену и презимену и разреду.
- Тестирати креирање објекта ове класе и приступ атрибутима.

```
package rs.math.oop.g08.p08.pristupPoljima;

public class Ucenik {
    String imePrezime;
    int razred;

    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
        prvi.imePrezime="Петар Перић";
        prvi.razred=3;
        Ucenik drugi = new Ucenik();
        drugi.imePrezime="Милан Микић";
        drugi.razred=6;
        System.out.println(prvi.razred);
        System.out.println(drugi.imePrezime);
    }
}
```

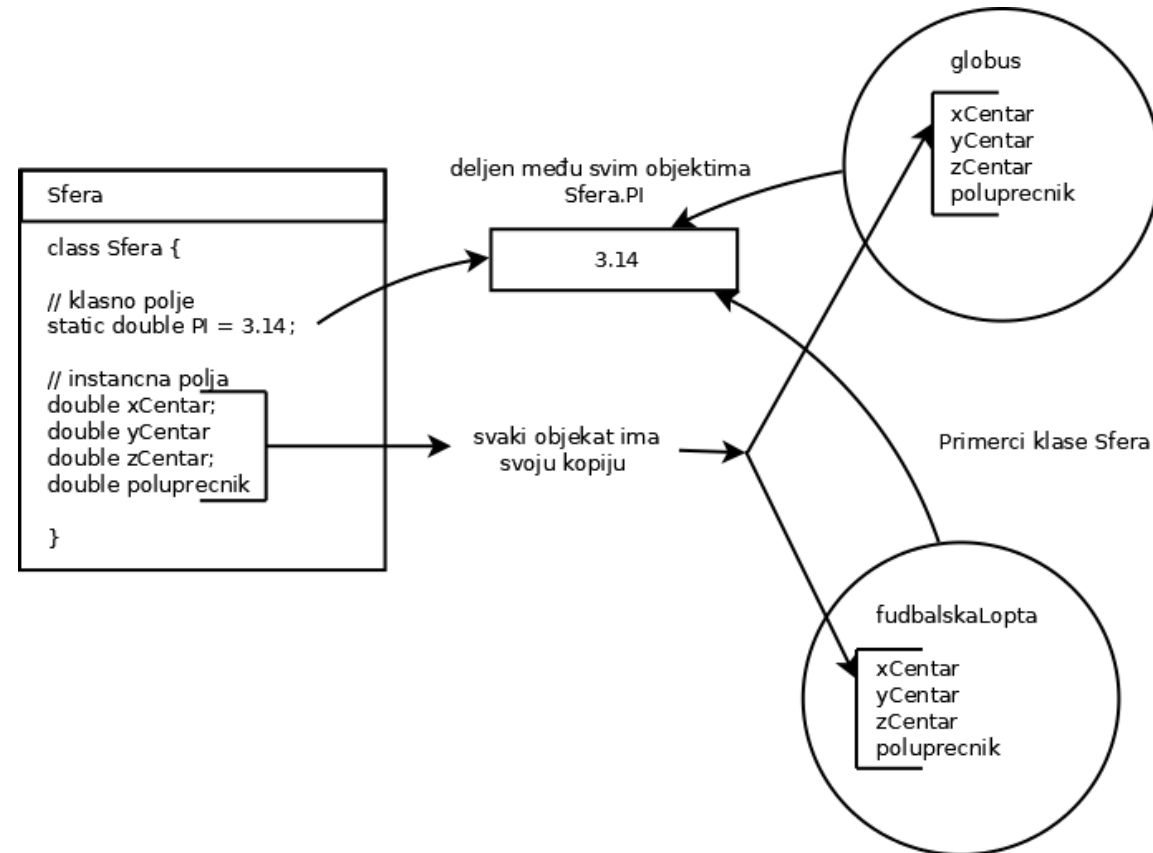
3

Милан Микић

СТАТИЧКА (КЛАСНА) ПОЉА

- Класа садржи само једну копију класног поља и то поље је дељено међу свим објектима дате класе.
- Класно поље постоји чак иако се не креира ниједан примерак дате класе.
- Класно поље се декларише коришћењем модификатора **static**.
- За приступ класној променљивој се користи тачка-нотација, при чему се као прималац поруке може користити:
 - име класе (препоручено)
 - или име неке инстанце класе.

СТАТИЧКА (КЛАСНА) ПОЉА (2)



ПРИМЕР 9

- Класу `Ucenik` из претходног примера проширити статичким пољем које у себи садржи број креираних инстанци класе `Ucenik`.
- Тестирати промену и читање вредности овог поља.

ПРИМЕР 9 (2)

```
package rs.math.oop.g08.p09.pristupStatickimPoljima;

public class Ucenik {
    String imePrezime;
    int razred;
    static int brojUcenika = 0;

    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
        prvi.imePrezime="Петар Перић";
        prvi.razred=3;
        prvi.brojUcenika++; // може се приступити преко инстанчне референце
        Ucenik drugi = new Ucenik();
        drugi.imePrezime="Милан Микић";
        drugi.razred=6;
        Ucenik.brojUcenika++; // али је преко назива класе природније
        System.out.println(prvi.brojUcenika);
    }
}
```

ОПСЕГ ВАЖЕЊА

- Опсег важења је део програма у којем променљива може да се користи.
- Код локалних променљивих, опсег важења је блок у којем је дефинисана од места где је дефинисана наниже.

```
{  
    int a=1; // овде се може приступити а, али не и b  
    { // овде се може приступити а, али не још b  
        int b=2; // овде се може приступити и а и b  
    } // овде се може приступити а, али не и b!  
}  
// овде се не може приступити нити а, нити b
```

ОПСЕГ ВАЖЕЊА (2)

- Класним пољима се може приступати у сваком тренутку рада програма, чак и пре него што је икоја инстанца припадајуће класе креирана.
- Пољима неког објекта се може приступати сво време док постоји тај објекат.

```
public static void main(String[] args) {  
    System.out.println(Ucenik.brojUcenika);  
    // грешка при компилацији, јер још не постоји објектна променљива  
    // System.out.println(prvi.imePrezime);  
    Ucenik prvi = null;  
    // грешка при извршавању, јер није још креиран објекат  
    //System.out.println(prvi.imePrezime);  
    prvi = new Ucenik();  
    prvi.imePrezime="Петар Перић";  
    prvi.razred=3;  
    System.out.println(prvi.imePrezime);  
}
```

МЕТОДИ

- Методи омогућавају да се дугачак код разбије у мање целине и на тај начин доприносе прегледности кода.
- Такође, метод се може позивати већи број пута што омогућава запис мање количине кода него у случају да методи нису подржани.
- Методи се појављују у телу класе и они се састоје из два кључна дела: потпис метода и тело метода.

```
class PripadajucaKlasa{  
    ...  
    povratni-tip imeMetoda(arg1, arg2, ..., argn){ // потпис метода  
        // код или тело метода  
    }  
    ...  
}
```


МЕТОДИ (2)

- Потпис два метода у класи мора бити различит, иначе компајлер не зна кога да позове.
- Да би метод вратио вредност, мора имати бар једну наредбу **return**.
 - Уколико метод не враћа вредност, његов тип је **void**.
- Аргументи метода се наводе у заградама иза имена метода (број аргумената може бити нула).
- Приликом декларације метода у класи, наводи се и тип за сваки аргумент. При позиву метода наводе се стварни аргументи.
- Тело метода може садржати декларације локалних променљивих и друге Јава наредбе.
- У наредбама у телу метода се могу користити четири потенцијална извора података:
 1. формални аргументи метода,
 2. инстанце и класне променљиве,
 3. локалне променљиве, дефинисане у телу метода и
 4. вредности које враћају други методи који су позвани у текућем.

ПОЗИВ МЕТОДА И КЉУЧНА РЕЧ `this`

- Слично пољима, инстанцни метод се може позвати тачка-нотацијом.
- Специјално, уколико се позива метод унутар метода истог објекта, није јасно како је могуће реферисати на променљиву.
- У том случају може се:
 - избећи употреба тачка-нотације;
 - употребити тачка-нотација над специјалном референтном вредношћу која се зове `this` и која представља референцу на текући објекат (у којем се налази позивајући метод).

ПРИМЕР 10

- Раније дефинисану класу **Ucenik** проширити методима за:
 - приказ информација о ученику
 - као и методу који закључује, на основу разреда, да ли ученик у старијој смени.

ПРИМЕР 10 (2)

```
package rs.math.oop.g08.p10.pozivanjeMetoda;

public class Ucenik {
    String imePrezime;
    int razred;

    void prikaziInformacije() {
        // позив метода jeStarijaSmena()
        // унутар другог метода који исто припада класи Ucenik
        System.out.println(imePrezime + " " + razred
            + " " + jeStarijaSmena()); // могло је и this.jeStarijaSmena()
    }

    boolean jeStarijaSmena() {
        return razred > 4;
    }
}
```

ПРИМЕР 10 (3)

```
package rs.math.oop.g08.p10.pozivanjeMetoda;

public class TestirajUcenik {

    public static void main(String[] args) {
        Ucenik prvi = new Ucenik();
        prvi.imePrezime = "Петар Перић";
        prvi.razred = 3;
        Ucenik drugi = new Ucenik();
        drugi.imePrezime = "Милан Микић";
        drugi.razred = 6;
        // позив преко тачка-нотације из метода main
        // који не припада класи Ucenik
        prvi.prikaziInformacije();
        drugi.prikaziInformacije();
    }
}
```

Петар Перић 3 false

Милан Микић 6 true

ПРЕОПТЕРЕЋЕЊЕ МЕТОДА

- Дефинисање метода са истим именом, али различитим аргументима назива се преоптерећење метода.
- У програмском језику С преоптерећивање није дозвољено.
- На пример, ако је потребно дефинисати методе које сабирају 2, 3 или 4 броја, дефинисала би се 3 метода са различитим: `saberi2(x, y)`, `saberi3(x, y, z)` и `saberi4(x, y, z, w)`.
- У Јави би се применом преоптерећивања то могло реализовати на следећи начин.

```
double saberi(double x, double y) { return x+y; }
```

```
double saberi(double x, double y, double z) { return x+y+z; }
```

```
double saberi(double x, double y, double z, double w) { return x+y+z+w; }
```

ПРИМЕР 11

- Дефинисати класу `Pravougaonik` описану координатама горњег левог и доњег десног темена.
- Претпоставка је да су странице правоугаоника паралелне са одговарајућим осама координатног система па је ова репрезентација јединствена.
- Користити целобројне координате имајући у виду матрицу пиксела на екрану.
- Омогућити подешавање ових координата на два начина:
 1. прослеђују су се два темена (било која, метод треба да процени да ли су валидна);
 2. прослеђује се само горње лево теме, ширина и висина.

ПРИМЕР 11 (2)

- Решење је предугачко за приказ на слајду. Погледати у књизи.

СТАТИЧКИ (КЛАСНИ) МЕТОДИ

- Статички методи су налик статичким пољима.
 - Они не припадају инстанци класе већ самој класи.
 - То значи да је и њихов животни циклус независан од постојања конкретне инстанце.
 - Попут функција у програмском језику C.
- На пример, метод `main(String args[])` је увек статички. Зашто?
- У Јави је употреба углавном за реализацију услужних класа.
 - На пример `Math`, `System` итд.
- Статички метод се разликује од обичног по основу модификатора `static`, који претходи повратној вредности у потпису метода.

СТАТИЧКИ МЕТОДИ (2)

```
int polje1;  
static int polje2;  
  
void metod1() { ... }  
  
static void metod2() { ... }  
  
static void metod3() {  
    // polje1 = 10; // не компајлира се  
    polje2 = 20;  
    // metod1(); // не компајлира се  
    metod2();  
}
```

КЛАСЕ И НАСЛЕЂИВАЊЕ

- За претходно креиране класе, могуће је креирати њихове поткласе помоћу кључне речи `extends`.
- Поткласа на тај начин постаје конкретизација своје надкласе, тако да:
 - садржи све наслеђене атрибуте и методе;
 - те их може користити или чак мењати;
 - такође, поткласа може додавати и нове атрибуте односно методе.
- Наслеђене атрибуте и методе је могуће користити на два начина у поткласи:
 1. применом тачка-нотације над референцом или
 2. позивањем унутар неког од метода поткласе.

ПРИМЕР 12

- Дефинисати класу **Srednjoskolac**, као поткласу класе **Ucenik**.
- Од додатних атрибута, средњошколац поседује и врсту средње школе.
- Поред тога, поседује и метод за узимање вредности атрибута врсте средње школе.

ПРИМЕР 12 (2)

```
public class Srednjoskolac extends Ucenik {
    String vrstaSkole;

    String uzmiVrstuSkole() {
        return vrstaSkole;
    }

    void proveriRazred() {
        if(razred>4 || razred<1)
            System.out.println("Разред "+razred+" није могућ у средњој школи.");
        else
            System.out.println("Разред "+razred+" је у реду.");
    }

    public static void main(String[] args) {
        Srednjoskolac sred = new Srednjoskolac();
        sred.imePrezime="Марко Родић";
        sred.razred=2;
        sred.vrstaSkole="Техничка школа";
        sred.prikaziInformacije();
        System.out.println(sred.vrstaSkole);
        System.out.println(sred.uzmiVrstuSkole());
        sred.proveriRazred();
        sred.razred=5;
        sred.proveriRazred();
    }
}
```

ПРИСТУПАЊЕ ПОЉИМА НАДКЛАСА

- Приступање пољима надкласа могуће је употребом кључне речи **super**.
- Ова кључна реч може (има смисла) да се позове само унутар инстанцног метода.
 - У случају да је метод статички није јасно шта је текући објекат па ни његова надкласа.

```
void proveriRazred() {  
    if(razred>4 || super.razred<1)  
        System.out.println("Разред "+razred+" није могућ у средњој школи.");  
    else  
        System.out.println("Разред "+razred+" је у реду.");  
}
```

- Кључна реч **super** омогућава приступ траженом пољу из директне или индиректне надкласе (све до класе **Object**).
 - Јава ће потражити поље најпре у директној надкласи, ако не нађе тражиће даље.
- **super** синтаксно подсећа на раније уведену кључну реч **this**.
 - Међутим, **super** није референца ка објекту надкласе (објекти надкласе се не креирају!).

САКРИВАЊЕ ПОЉА

- Сакривање поља (енг. field hiding) је постојање два или више истоимених поља у оквиру хијерархије наслеђивања конкретне класе.
- Потребно је користити `super` како би се разрешили неједнозначност.
- Није препоручено сакривати поља!

```
class A{
    String naziv;
}

class B extends A{
    String naziv;

    void prikazi(){
        System.out.println(naziv); // назив из класе B
        System.out.println(super.naziv); // назив из класе A
    }
}
```

ПОЗИВАЊЕ МЕТОДА НАДКЛАСА

- Аналогно приступању пољима надкласа.
- Да би се приступило понекад је довољно само навести назив метода.
 - Ово се односи на ситуацију када класа не садржи истоимени метод.

```
class A{
    void prikazi(){ ... }
}

class B extends A{ ... }

class C extends B{
    void metod(){
        prikazi(); // позив метода дефинисаног у А
        super.prikazi(); // исти ефекат
    }

    static void main(String[] args){
        C c = new C();
        c.prikazi(); // подразумева позив метода дефинисаног у А
    }
}
```


ПОЗИВАЊЕ МЕТОДА НАДКЛАСА (2)

- Ако класа садржи истоимени метод, потребно је користити кључну реч **super**.

```
class C extends B{
    void prikazi(){ ... }

    void metod(){
        prikazi(); // позив метода дефинисаног у С
        super.prikazi(); // позив метода дефинисаног у А
    }

    static void main(String[] args){
        C c = new C();
        c.prikazi(); // подразумева позив метода дефинисаног у С
    }
}
```

ПРЕВАЗИЛАЖЕЊЕ МЕТОДА

- Превазилажење или редефинисање метода је ситуација у којој метод класе поново дефинише метод који већ постоји у некој његовој надкласи.
- Претходна дефиниција метода и даље постоји, али јој се сада обавезно приступа уз употребу кључне речи **super**.
 - Како би се у коду нагласило да је реч о превазилажењу, а не о првој дефиницији метода са датим називом, користи се специјална мета-језичка ознака **@Override**.

```
class C extends B{
    @Override
    void prikazi(){ ... }

    void metod(){
        prikazi(); // позив метода дефинисаног у C
        super.prikazi(); // позив метода дефинисаног у A
    }

    static void main(String[] args){
        C c = new C();
        c.prikazi(); // подразумева позив метода дефинисаног у C
    }
}
```

ПРЕВАЗИЛАЖЕЊЕ МЕТОДА (2)

- Када компајлер прочита анотацију `@Override` он ће проверити да ли метод са датим називом већ постоји у некој од надкласа.
 - Ако не постоји, пријавиће грешку, јер није могуће превазићи метод који не постоји.
 - Постојање ове анотације смањује евентуалне грешке у којима програмер бива уверен да је редефинисао неки метод, а уствари га по први пут дефинише, нпр. због словне грешке.
- Постоји много разлога због којих би програмер превазишао метод надкласе. Наводимо у наставку примере три најчешће превазиђена, сва три из класе `Object`:
 - `toString()` метод за текстуални приказ објекта;
 - `equals()` метод за поређења два објекта на једнакост;
 - `hashCode()` метод који реализује хеш-функцију, тј. враћа целобројну репрезентацију датог објекта.

ПРЕВАЗИЛАЖЕЊЕ toString()

```
public class Ucenik {
    String imePrezime;
    int razred;

    @Override
    public String toString() {
        return imePrezime+" "+razred;
    }
}

public class Srednjoskolac extends Ucenik {
    String vrstaSkole;

    @Override
    public String toString() {
        return super.toString()+" "+vrstaSkole;
    }
}
```

ПРЕВАЗИЛАЖЕЊЕ equals()

- Раније је било речи о неадекватности употребе оператора == за поређење објеката.
- Метод equals() треба да дефинише суштинско поређење, а не поређење референци.
- Испод је дат пример реализације метода equals() у оквиру класе String.

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true; // иста референца, дакле исти и садржај
    if (!(obj instanceof String))
        return false; // аргумент није String па сигурно не може бити исти
    String sObj = (String) obj;
    if (length() != sObj.length())
        return false; // сигурно нису исти, јер имају различиту дужину
    for (int i = 0; i < length(); i++) {
        if (charAt(i) != sObj.charAt(i))
            return false; // прво непоклапање, значи различити су
    }
    return true; // није било непоклапања, значи да су исти
}
```

ПРЕВАЗИЛАЖЕЊЕ hashCode()

- Хеш-код представља целобројну репрезентацију објекта.
- Хеш-код није нужно јединствен, тј. може постојати више објеката који имају исти.
 - Али је овакве ситуације треба минимизовати добром реализацијом `hashCode()`.
- Да би се конзистентно дефинисала хеш функција (`hashCode()`) за неки објекат, програмер мора редефинисати две методе: `hashCode()` и `equals()`.
 - Мора бити испуњено да ако важи `a.equals(b)`, онда `a` и `b` морају имати исти хеш-код.
- Пример превазиђеног `hashCode()` за класу `String` је дат испод.

```
public int hashCode() {  
    int h = 0;  
    for (byte v : value) { -  
        h = 31 * h + (v & 0xff);  
    }  
    return h;  
}
```

ПРИМЕР 13

- Раније уведена класа **Таска**, која представља тачку у дводимензионалном простору целобројних координата, прилагодити тако да се у класи превазилази подразумевано понашање метода **toString()**, **equals()** и **hashCode()**.
- Потом дефинисати класу **Duz**, описану са две тачке, и за њу такође редефинисати поменуте методе.

ПРИМЕР 13 (2)

- Решење је предугачко за слајдове, налази се у тексту поглавља.

ПОДЕШАВАЊЕ ПОЧЕТНОГ СТАЊА ОБЈЕКТА

- Након што се објекат креира на хипу, могуће је подесити његово почетно стање, односно вредности припадајућих поља.
- Уколико се не наведе подешавање стања примениће се подразумеване вредности.
- Могуће је подесити стање применом следећа четири механизма:
 - иницијализација поља при декларацији (било да је статичко или не);
 - иницијализација инстанчних поља у иницијализационом блоку;
 - иницијализација статичких поља у статичком иницијализационом блоку;
 - иницијализација поља у оквиру конструктора (у конструктору се могу вршити и остале активности, а не само подешавање поља).
- Прво се врше статичке иницијализације, па инстанчне и онда тек конструктори.

ИНИЦИЈАЛИЗАЦИОНИ БЛОК

- Иницијализациони блок служи да се у оквиру њега иницијализују вредности променљивих, тачније поља.
- За разлику од иницијализације приликом декларације променљиве, у оквиру иницијализационог блока је могуће користити контролне структуре.

```
static int n;  
int suma;  
  
static {  
    n = 10;  
}  
  
{  
    suma = 0;  
    Random rg = new Random();  
    for(int i=0; i<n; i++)  
        suma+=rg.nextInt();  
}
```

КОНСТРУКТОР

- Приликом креирања конкретног примерка неке класе позива се конструктор.
- Конструктор се у функционалном смислу може сврстати у методе. Разлике су:
 - конструктор се имплицитно позива при позиву оператора **new**;
 - конструктор нема повратну вредност.

```
class Tacka{
    int x, y;

    public Tacka(){
        x = 10;

        y = 100;
    }

    public static void main(String[] args){
        Tacka t = new Tacka();
    }
}
```

ПОДРАЗУМЕВАНИ КОНСТРУКТОР

- Уколико програмер није дефинисао конструктор за дату класу, преводилац позива подразумевани конструктор.
- Подразумевани конструктор нема аргумената, иницијализује све инстанцне и класне променљиве на подразумеване вредности.

```
class Tacka{  
    int x, y;  
  
    public static void main(String[] args){  
        Tacka t = new Tacka();  
    }  
}
```

ПОДРАЗУМЕВАНИ КОНСТРУКТОР (2)

- Ако је програмер дефинисао бар један конструктор за дату класу, онда подразумевани конструктор више не постоји.

```
class Tacka{
    int x, y;

    public Tacka(int xp, int yp){
        x = xp;
        y = yp;
    }

    public static void main(String[] args){
        // Tacka t = new Tacka(); // произвело би грешку
        Tacka t = new Tacka(10, 20);
    }
}
```

ПРЕОПТЕРЕЋЕЊЕ КОНСТРУКТОРА

- Конструктори могу бити преоптерећени, исто као и остали методи.
- Ако постоји додатни конструктор, који има неке нове особине, у њему се може позвати већ постојећи конструктор употребом кључне речи **this**.

ПРИМЕР 14

- Прилагодити класу **Таска** тако да садржи два конструктора, један без аргумената, други са вредностима за **x** и **y** координате.
- Потом преоптеретити и конструкторе за класу **Duz**.
- Потребно је да постоји празан конструктор, конструктор који прихвата две тачке и конструктор који прихвата четири цела броја (редом координате тачака).

ПРИМЕР 14

- Решење је предугачко за слајдове, налази се у тексту поглавља.

РЕФЕРЕНЦИЈАЛНА ЗАВИСНОСТ ОБЈЕКТА

- Објекат чије унутрашње стање (тј. вредност неког поља) може да се промени назива се мутирајући објекат.
- У супротном се ради о немутирајућем објекту.
 - Стога треба пажљиво радити са конструкторима мутирајућих објеката.
- Наиме, може се догодити да се, при извршавању конструктора, вредност поља новокреираног објекта постави тако да садржи вредност аргумента конструктора:
 - у том случају стварни аргумент конструктора и поље новокреираног објекта постају „везани“ и реферишу на исти објекат (ово се још зове и референцијална зависност);
 - због тога промена објекта аргумента конструктора доводи до промене поља новокреираног објекта и обрнуто.

РЕФЕРЕНЦИЈАЛНА ЗАВИСНОСТ ОБЈЕКТА (2)

```
Tacka t1 = new Tacka(2, 3);
Tacka t2 = new Tacka(4, 5);
Duz d = new Duz(t1, t2);
System.out.println(d); // {(2, 3), (4, 5)}
t1.x = 20;
System.out.println(t1); // (20, 3)
System.out.println(d); // {(20, 3), (4, 5)}
```

- Тачка `t1` описује дуж `d`.
- Након што се тачка `t1` измени (промена `x` координате), измена се рефлектује и на унутрашње стање дужи `d`.
- Разлог томе је дефиниција конструктора дужи:

```
public Duz(Tacka a, Tacka b){
    this.a = a;
    this.b = b;
}
```

КОПИРАЈУЋИ КОНСТРУКТОР

- Претходно описано понашање често није пожељно, јер нарушава принцип учауривања података у оквиру објекта.
- Како би се разрешио проблем референцијалне зависности користи се копирајући конструктор (или конструктор копије).
- На пример, копирајући конструктор за класу **Таска** може бити реализован овако:

```
public Таска(Таска t){  
    x = t.x; // на нивоу примитивних типова нема референцијалне зависности  
    y = t.y;  
}
```

- Даље, конструктор за класу **Duz** би се дефинисао на следећи начин:

```
public Duz(Duz d){  
    a = new Таска(d.a); // овим се елиминише референцијална зависност  
    b = new Таска(d.b);  
}
```

ПРИМЕР 15

- Нека је цртеж у векторској графици представљен низом од највише 10 полигона.
- Полигон је описан низом од највише 10 тачака.
- Тачка је, као и у претходним примерима, представљена са две целобројне координате.
- Креирати копирајући конструктор за цртеж.
- Да би се ово постигло потребно је направити копирајуће конструкторе за сваку од наведених класа.
- Свака класа сама брине о креирању сопствене копије.

ПРИМЕР 15 (2)

- Решење је предугачко за слајдове, налази се у тексту поглавља.

ПОЗИВ КОНСТРУКТОРА НАДКЛАСЕ

- Понекад је је понашање конструктора класе надкуп понашања конструктора надкласе, што представља проблем, јер не желимо да дуплирамо код.
- Могуће је позвати и конструктор надкласе наредбом `super(lista argumenata)`.

ПРИМЕР 16

- Дефинисати поткласу класе `Taska` под називом `OznacenaTaska`.
- Ова класа садржи и поље `oznaka` типа `String`.
- Приликом дефиниције конструктора класе `OznacenaTaska` искористити раније дефиниције конструктора класе `Taska`.

ПРИМЕР 16 (2)

```
public class OznacenaTacka extends Tacka{
    String oznaka;

    public OznacenaTacka() {
        super();
        oznaka = "";
    }

    public OznacenaTacka(int x, int y, String oznaka) {
        super(x, y);
        this.oznaka = oznaka;
    }

    @Override
    public String toString() {
        return oznaka+super.toString();
    }

    public static void main(String[] args) {
        OznacenaTacka ot = new OznacenaTacka(10, 20, "A");
        System.out.println(ot);
        Tacka t = new OznacenaTacka(30, 40, "B");
        System.out.println(t);
    }
}
```


МОДИФИКАТОРИ ВИДЉИВОСТИ

- Модификатори видљивости су специјалне кључне речи које мењају понашање класа, метода и поља.
- Постоје четири нивоа видљивости, тзв. “4P – заштита”:
 - `public`
 - `protected`
 - `package` (имплицитни – не записује се)
 - `private`

МОДИФИКАТОР `public`

- Модификатор `public` даје пољу/методу/класи најширу (јавну) видљивост.
 - То значи да се њима може приступити са било које локације.

```
public class A {  
    public int polje;  
    public A() { }  
    public void metod() {  
        this.polje = 10;  
    }  
    void testiraj() {  
        polje = 10;  
        metod();  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }  
    public static void main(String[] args) {  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }  
}
```

МОДИФИКАТОР `public` (2)

```
package rs.math.oop.g08.p17.publicModifikator.podpaket1;
```

```
public class B {
```

```
    void testirajA() {  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }
```

```
    public static void main(String[] args) {  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }  
}
```

МОДИФИКАТОР `public` (3)

```
package rs.math.oop.g08.p17.publicModifikator.podpaket2;
import rs.math.oop.g08.p17.publicModifikator.podpaket1.A;

public class C {

    void testirajA() {
        A a = new A();
        a.polje = 20;
        a.metod();
    }

    public static void main(String[] args) {
        A a = new A();
        a.polje = 20;
        a.metod();
    }
}
```

МОДИФИКАТОР package

- Модификатор package је имплицитни.
- Видљивост је, применом овог модификатор, ограничена на пакет у којем је класа.

```
package rs.math.oop.g08.p18.packageModifikator.podpaket1;
```

```
class A {  
    int polje;  
    A() { }  
    void metod() {  
        this.polje = 10;  
    }  
    void testiraj() {  
        polje = 10;  
        metod();  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }  
    public static void main(String[] args) {  
        A a = new A();  
        a.polje = 20;  
        a.metod();  
    }  
}
```

МОДИФИКАТОР package (2)

- Иста је ситуација и у оквиру тела класе B.
- Класу C међутим није могуће компајлирати.

```
package rs.math.oop.g08.p18.packageModifikator.podpaket2;

// коментарисани редови су разлог немогућности компилације
// import rs.math.oop.g08.p18.packageModifikator.podpaket1.A;

public class C {

    void testirajA() {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }

    public static void main(String[] args) {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
    }
}
```

МОДИФИКАТОР `protected`

- Модификатор `protected` подразумева видљивост коју обезбеђује и `package`.
- Додатно `protected` омогућава видљивост и ван пакета уколико класа из које приступамо наслеђује класу којој се приступа.
 - У овом случају поља и методи којима приступамо нису чланови класе А већ изведене класе С.

```
package rs.math.oop.g08.p19.protectedModifikator.podpaket1;
public class A {
    protected int polje;
    protected A() { }
    protected void metod() { this.polje = 10; }
    void testiraj() {
        ...
    }
    public static void main(String[] args) {
        ...
    }
}
```

МОДИФИКАТОР `protected` (2)

- С обзиром да `protected` подразумева и `package` код класе **B** је ситуација иста као и раније.
- Код класе **C** је ситуација сада доста другачија.

```
package rs.math.oop.g08.p19.protectedModifikator.podpaket2;
import rs.math.oop.g08.p19.protectedModifikator.podpaket1.A;
public class C extends A{
    void testirajA() {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
        polje = 20;
        metod();
    }
    public static void main(String[] args) {
        // A a = new A();
        // a.polje = 20;
        // a.metod();
        C c = new C();
        c.polje = 20;
        c.metod();
    }
}
```


МОДИФИКАТОР `private`

- Модификатор `private` нуди најрестриктивнију видљивост.
- Видљивост је употребом овог модификатора ограничена на тело класе.
 - Стога класе **B** и **C** сада више није могуће компајлирати.

```
package rs.math.oop.g08.p20.privateModifikator.podpaket1;
class A {
    private int polje;
    private A() { }
    private void metod() { this.polje = 10; }
    void testiraj() {
        ...
    }
    public static void main(String[] args) {
        ...
    }
}
```

ЗАШТИТА ПОЉА ПРИМЕНОМ МОДИФИКАТОРА `private`

- Модификатор `private` има значајну улогу у реализацији усауривања поља, јер онемогућава неконтролисани приступ пољима из “спољног света”.
- Када је реч о приступу методима, за њих се користи модификатор `private` кад год је улога метода стриктно локална.
- Како онда користити поља која су приватна ван тела класе?

```
public class A{  
    private int polje;  
}
```

МЕТОДИ ЗА УЗИМАЊЕ И ПОСТАВЉАЊЕ ВРЕДНОСТИ

- Одговор је: помоћу метода за узимање и постављање вредности (енг. getter и setter методи).

```
public class A{
    private int polje;

    public int uzmiPolje(){
        return polje;
    }

    public void postaviPolje(int polje){
        this.polje = polje;
    }
}
```

```
A a = new A();
a.postaviPolje(10);
System.out.println(a.uzmiPolje());
```

МЕТОДИ ЗА УЗИМАЊЕ И ПОСТАВЉАЊЕ ВРЕДНОСТИ (2)

- Зашто је то боље од варијанте која користи јавно поље?

```
public class A{  
    public int polje;  
}
```

- Варијанта са јавним пољем је краћа и нуди идентичну функционалност:
 - поље `polje` је могуће прочитати из било ког дела кода;
 - пољу `polje` је могуће променити вредност из било ког дела кода.

```
A a = new A();  
a.polje = 10;  
System.out.println(a.polje);
```

- Приватна варијанта боље чува унутрашње стање објекта.
- Такође омогућава селективни приступ само за писање или само за постављање.

ПРИМЕР 21

- Дефинисати класу **Krug** која је описана центром и полупречником.
- Додатно, ова класа садржи и поље **obim** и **povrsina**.
- Потребно је дефинисати конструктор који прихвата као аргументе центар и полупречник.
- Такође редефинисати метод **toString()**.
- Реализовати две варијанте ове класе:
 - једну која не користи заштиту поља,
 - и другу која користи.
- Тестирати и упоредити употребу ове две варијанте

ПРИМЕР 21 (2)

- Решење је предугачко за слајд, погледати у књизи.

МОДИФИКАТОР ОГРАНИЧАВАЊА - **final**

- Модификатор **final** се користи за ограничавање понашања класа, поља и метода:
 - **final** класе не могу бити наслеђене;
 - **final** поља не могу променити вредност након иницијализације – ефективно постају константна;
 - **final** методи не могу бити редефинисани.

СПРЕЧАВАЊЕ НАСЛЕЂИВАЊА

- Пример **final** класе је **String**. Због тога се следећи код не компајлира:

```
public class MojString extends String{  
    ...  
}
```

- Дефиниција класе **String** има следећу форму:

```
public final class String ...{  
    ...  
}
```


КОНСТАНТНА ПОЉА

- Вредност поља је могуће одржати константном употребом модификатора **final**.
- Да би се постигло овакво понашање компајлер анализира изворни код и утврђује да ли се додела вредности пољу (оператор =) применила сам једном.
 - Ако у коду постоји више додела, компајлер пријављује грешку.
- Испод је дат пример константног и притом статичког поља **Math.PI**.

```
...  
private void preracunaj() {  
    p = r*r*Math.PI;  
    o = 2*r*Math.PI;  
}  
...
```

СПРЕЧАВАЊЕ РЕДЕФИНИСАЊА

- Превазилажење (редефинисање) метода је моћан концепт.
 - Омогућава прилагођавање понашања изведених класа.
- Понекад је ову могућност, ипак, потребно спречити.
 - На пример, ако смо сигурни да је дефиниција метода у текућој класи тачна за све евентуалне поткласе.

```
public class Kvadrat {  
    private double a;  
    ...  
    final double površina() {  
        return a*a;  
    }  
}
```

ПИТАЊА И ЗАДАЦИ

- Биће накнадно додато...