Objektno orijentirano programiranje

3: Definiranje klasa i prava pristupa. Konstruktori. Varijabilni broj argumenata. Statičke metode i varijable

Creative Commons

You are free to

- **Share** — copy and redistribute the material in any medium or format
- **Adapt** — remix, transform, and build upon the material

under the following terms

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
- **NonCommercial** — You may not use the material for commercial purposes.
- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

- https://creativecommons.org/licenses/by-nc-sa/4.0/

Kategorizacija

- Kategorizacija: razvrstavanje stvari u grupe, tj. klase
- Program koji se razvija odnosi se na neku (poslovnu) domenu
- Svaka domena ima
  - stvarne entitete koji se moraju kategorizirati (podijeliti u klase)
    - npr. fakultet: studenti, nastavnici, predmeti, ispiti, …
    - entiteti imaju atribute: ime, prezime, naziv predmeta, ocjena, …
  - postupke vezane uz pojedinu identificiranu kategoriju
    - npr. upis predmeta, prijava ispita, izračun ocjene, …
    - bit će modelirani kao metode pojedinih klasa
  - i „poslovna” pravila
    - npr. „za ocjenu 5 treba imati minimalno 90% bodova”, …
    - bit će ugrađenu u postupke (metode)

Apstrakcija u objektno-orijentiranom programiranju

- Identificirani entiteti i akcije moraju se modelirati kao klase i metode
- Apstrakcija: modeliranje entiteta tako da se koriste samo bitne komponente stvarnog entiteta

- Što je bitno za npr. modeliranje studenta?
  - ime, prezime i JMBAG su vjerojatno bitni

  - datum rođenja, mentor, visina, težina, … ? - ovisno o namjeni sustava, tj. što modeliramo
    - radimo li sustav za organiziranje sportskih događaja?
    - …ili možda sustav za evidenciju studija i izdavanje potvrda?

Učahurivanje (Enkapsulacija)

- Kako radi TV?
  - sve dok radi i dok imamo sučelje za upravljanje TV-om nije bitno
- Hoćemo li studentove ocjene spremati u polju ili negdje drugdje? Je li nam bitno kako se zove varijabla koja sadrži ocjene?
  - želimo li dopustiti bilo kome da mijenja podatke direktno ili isključivo putem ponuđenih metoda?
  - potrebno je zaštititi implementacijske detalje od javnog pristupa kako bi se izbjegle slučajne ili namjerne pogreške
- Enkapsulacija (učahurivanje)
  - veže podatke i metode koji rade nad tim podacima
  - skriva implementacijske detalje i sprječava direktni pristup
  - pridonosi slaboj povezanost objekata (engl. *loose coupling*)
    - slabom povezanošću objekti postaju neovisniji i interne promjene jednog objekta ne utječu na rad drugog

Modifikatori vidljivosti u Javi

- Metode klase uvijek mogu pristupati članskim varijablama i metodama svoje klase* **o (ne)statičkim članovima naknadno*
- Modifikatori vidljivosti ograničavaju pristup (vidljivost, dostupnost) iz koda u ostalim klasama
  - *private:* pristup nije dozvoljen (tj. moguć samo iz koda iste klase)
  - *public:* javno izloženo, dostupno svima
  - *protected*
    - pristup moguć iz koda bilo koje klase u istom paketu ili iz naslijeđenog objekta (o klasama i nasljeđivanju u 4. temi)
  - bez eksplicitno navedenog modifikatora vidljivosti
    - tzv. *package-private* vidljivost
    - pristup moguć iz koda klasa koje pripadaju istom paketu
- Vršni objekti (npr. klase) mogu biti *public* ili *package-private*
  - članovi (varijable, metode, ugniježđene klase) mogu imati i ostale modifikatore

Razlike u odnosu na neke druge jezike

- Ključna razlika je u značenju modifikatora *protected* i situacije kad modifikator vidljivosti nije naveden

Modifikator vidljivosti	Java	C#	C++
public	(isto značenje u Javi C#-u i C++u)		
private		(isto značenje)	
protected	Kod u klasama iz istog paketa ili iz naslijeđenih objekata	iz naslijeđenih objekata	iz naslijeđenih objekata
internal protected	-	Sve klase unutar istog projekta (*assembly*) i potklase	-
private protected	-	Potklase ali samo ako su iz istog projekta	-
internal	-	Klase iz istog projekta (*assembly*)	-
	Klase iz istog paketa	(=private)	(=private)

Apstrakcija točke koordinatnog sustava

- Točka određena koordinatama (x, y)
  - 2 atributa (članske varijable) tipa *double*
    - varijable će biti privatne te će postojati tzv. *getteri* i *setteri* za dohvat i promjenu vrijednosti učahurenih vrijednosti
- Metode:
  - *print()* za ispis točke na standardni izlaz
    - naknadno će biti zamijenjeno s metodom *toString()*
  - *isEqualTo(Point other)* za usporedbu s nekom drugom točkom
    - naknadno će biti zamijenjeno s metodom *equals(Object obj)*

Učahurivanje u klasi *Point*

- *Getter* i *setter* uobičajeno imaju nazive *[get/set]ImeVarijable*
  - U Javi se za metode uobičajeno koristi *camelCase* methods

```
package hr.fer.oop.constructors
public class Point {
    private double x, y;
    public double getX(){
       return x;
    }
    public void setX(double x){
       this.x = x;
    }
    public double getY(){
       return y;
    }
    public void setY(double y){
       this.y = y;
    }   ...
```

03_Constructors/…/hr/fer/oop/constructors/Point.java

Usporedba točaka (1)

- Dvije točke ćemo smatrati jednakim ako su im koordinate jednake

03_Constructors/…/ hr/fer/oop /constructors/Point.java

```
public class Point {
    ...
    public void print(){
       System.out.printf("(%.2f, %.2f)%n" , x , y);
    }

    public boolean isEqualTo(Point other) {
       return x == other.x && y == other.y;
    }
```

- Napomena: prikazano rješenje nije dobro, jer uspoređuje realne brojeve koristeći operator == te je moguće neispravno ponašanje zbog numeričke nepreciznosti
  - npr. 3 * 0.1 == 0.3 nije istina
  - 3 * 0.1 je 0.3000000000000004, a 0.3 ima beskonačan broj binarnih znamenki. Stoga je $3 * 0.1 - 0.3 \approx 5.55 * 10^{-17}$

Usporedba točaka (2)

- Bolji pristup je provjeriti je li razlika između 2 realna broja relativno dovoljno mala (npr. 0.001% jednog od brojeva)
  - za primjere koji slijede 10^{-8} će biti sasvim dovoljno

03_Constructors/…/hr/fer/oop /constructors/Point.java

```
public class Point {
    private double x, y;
    ... getters and setters ...

    public void print(){
       System.out.printf("(%.2f, %.2f)%n" , x , y);
    }

    public boolean isEqualTo(Point other) {
       return Math.abs(x-other.x) < 1E-8
              &&
            Math.abs(y-other.y) < 1E-8;
    }
```

Kreiranje objekta tipa Point

- Novu točku stvaramo (instanciramo) korištenjem operatora new

```
Point p = new Point();
```

  - vrijednosti varijabli x i y u tako stvorenom objektu su 0 (pretpostavljane vrijednosti za tip podatka double)
  - naknadno se mogu promijeniti s *p.setX(new_value)* i *p.setY(value)*
- Što ako vrijednosti x i y treba postaviti na neku vrijednost odmah prilikom stvaranja objekta tipa Point?
  - može im se pridružiti vrijednost odmah pri deklaraciji

```
private int x = 5;
```

  - može se napisati jedan ili više konstruktora i postaviti vrijednost u kodu konstruktora
  - Napomena: prvo se redom postavljaju vrijednosti navedene u deklaraciji, a onda se ide na izvršavanje konstruktora

Konstruktor

- Posebna metoda koja služi za inicijalizaciju novog objekta (npr. pridruživanje vrijednosti članskim varijablama)
  - ne može se direktno pozvati
    - izvršava se odmah nakon rezervacije memorije i stvaranja objekta operatorom *new*
  - naziv konstruktora isti kao i naziv klase
  - može imati argumente, ali nema povratni tip (čak ni void)
- Klasa može imati nula ili više eksplicitno napisanih konstruktora
  - ako nije napisan niti jedan konstruktor, onda je Javin prevodilac automatski stvara tzv. podrazumijevani konstruktor (bez parametara)
    - u slučaju kad je napisan barem jedan konstruktor, podrazumijevani konstruktor se neće automatski stvoriti

Konstruktor za klasu *Point*

- Konstruktor s dva argumenta: vrijednosti koje treba upotrijebiti kao vrijednosti za *x* i *y*.

```
public class Point {
  private double x, y;
  public Point(double newX, double newY){
        x = newX;
        y = newY;
  }
  ...
}
```

- Nova točka se sad stvara s npr. `Point p = new Point(2.0, 5);`
  - Kod `Point p = new Point();` više nije ispravan, jer ne postoji konstruktor bez argumenata
    - Javin prevodilac ga nije stvorio, jer je napisan barem jedan konstruktor
    - ako je potrebno, možemo sami napisati konstruktor bez argumenata

Skrivanje varijabli i ključna riječ *this*

- Što ako je naziv argumenta jednak nazivu članske varijable?
  - uobičajeno u konstruktorima, getterima i setterima
  - npr. što ako se argumenti u konstruktoru promijene iz *newX* i *newY* u *x* i *y*?
- Ime se odnosi na argument (čime se skriva članska varijabla), a člansku varijablu možemo dobiti pomoću reference na trenutni objekt: *this*

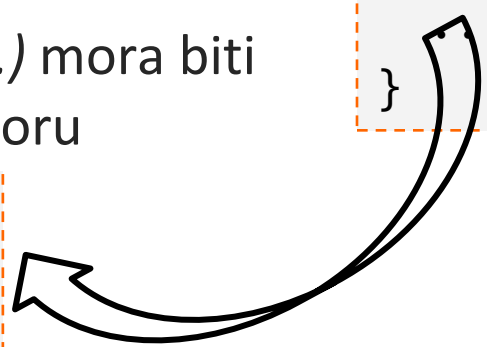
```
public class Point {
  private double x, y;
  public Point(double x, double y){
        this.x = x;
        this.y = y;
  }
  ...
}
```

this za poziv nekog drugog konstruktora

- Dodatni konstruktor koji inicijalizira vrijednosti točke na osnovu neke druge točke
  - ponavlja se (gotovo) isti kod
- Elegantnije rješenje s *this(argumenti)*
  - izvršava kod iz postojećeg konstruktora tj. poziva postojeću metodu
  - ako se želi koristiti, *this(…)* mora biti prva naredba u konstruktoru

```
public class Point {
  private double x, y;
  public Point(Point p) {
        x = p.x;
        y = p.y;
  }
  ...
}
```

```
public class Point {
  private double x, y;
  public Point(Point p) {
        this(p.x, p.y);
  }
  ...
}
```



Napomena: *this* ne stvara novi objekt

- novi objekti nastaju operatorom *new*

Primjer s korištenjem različitih konstruktora

- Što ispisuje sljedeći program?

```
package hr.fer.oop.constructors;
public class Main {
   public static void main(String[] args) {
       Point p1 = new Point(2, 5);
       Point p2 = new Point(p1);
       System.out.println("p1.isEqualTo(p2) : "
           + p1.isEqualTo(p2)); //true or false?
       p1.setX(1);
       p1.setY(2);
       System.out.println("p1.isEqualTo(p2) : "
           + p1.isEqualTo(p2)); //true or false?

       p1.print();
       p2.print();
     ...
```

```
(1.0, 2.0)  or  (1.0, 2.0)  or  (2.0, 5.0)
(2.0, 5.0)      (1.0, 2.0)      (2.0, 5.0)
```

03_Constructors/…/hr/fer/oop/constructors/Point.java

Statičke metode

- Metode *print* i *isEqualTo* su metode objekata tipa *Point*
  - da bi se metoda mogla pozvati, mora postojati objekt nad kojim se izvršava
    - poziv oblika `nekaTocka.nazivMetode(argumenti)`
    - koristi podatke objekta (u ovom slučaju x i y) i druge raspoložive metode
- Metode se mogu označiti statičkim
  - za takve metode ne treba postojati primjerak te klase da bi se metoda mogla pozvati
    - pozivaju se s `NazivKlase.nazivMetode(argumenti)`
    - ne pripadaju pojedinom objektu, već klasi
    - ne mogu koristiti ne-statička polja i ne-statičke metode te klase
  - Napomena: Java dopušta da se statička metoda pozove i nad objektom te klase `object.method(arguments)`, ali to ne bi trebalo prakticirati (besmisleno je)

Primjer statičke metode

- Napisati metodu koja stvara novu točku koja predstavlja težište tri postojeće točke referencirane varijablama *a, b, c*
- Pretpostavimo da je kreirana metoda objekta (engl. *instance method*) u klasi *Point*
  - metodu bi pozivali s *a.centerOf(b, c)* ili nekom od permutacija varijabli *a, b,* i *c*
  - nema smisla – metoda nije namijenjena da bude dio jednog objekta
    - jednako pripada trima navedenim objektima
    - sličan primjer je *Integer.parseInt("12")* koji ne zahtijeva da postoji neki objekt tipa *Integer* prije parsiranja *stringa*
- Metoda pripada klasi, bit će označena sa *static* i pozivat će se s *Point.centerOf(a, b, c)*

Statička metoda za težište triju točaka (1)

- Metoda *centerOf* je označena sa *static*
  - navodi se između modifikatora vidljivosti i povratnog tipa
- U konkretnom primjeru, statička metoda kreira novu točku

03_Constructors/…/hr/fer/oop/staticmethods/Point.java

```
package hr.fer.oop.staticmethods;
public class Point
  ...
  public static Point centerOf(Point a, Point b, Point c) {
      double x = (a.x + b.x + c.x) / 3.;
      double y = (a.y + b.y + c.y) / 3.;
      Point p = new Point(x, y);
      return p;
  }
  ...
```

Statička metoda za težište triju točaka (2)

- Method *centerOf* je statička metoda klase *Point*
- Method *print* je metoda objekta (engl. *instance method*) tipa *Point*

```
package hr.fer.oop.staticmethods;
public class Main {

	public static void main(String[] args) {
		Point a = new Point(0,0);
		Point b = new Point(6,0);
		Point c = new Point(3,5);
		Point center = Point.centerOf(a, b, c);
		center.print();
		...
```

03_Constructors/…/hr/fer/oop/staticmethods/Main.java

Težište polja točaka (1)

- Klasa može imati više metoda istog imena, sve dok imaju različiti broj argumenata ili različite argumente
  - ovaj koncept se naziva preopterećivanje/višeznačnost (eng. ***overloading***)
  - druga metoda imena *centerOf* prima polje točaka

```
package hr.fer.oop.staticmethods;
public class Point
  …
  public static Point centerOf(Point[] points) {
    double x = 0, y = 0;
    int len = points.length;
    for(int i=0; i<len; i++){
      x += points[i].x;       y += points[i].y;
    }
    Point p = new Point(x / len, y / len);
    return p;
  }  ...
```

03_Constructors/…/hr/fer/oop/staticmethods/Point.java

Težište polja točaka (2)

- Umjesto klasične for-petlje može se koristiti tzv. for-each varijanta
  - iterira kroz polje točaka i u svakom koraku adresu trenutne točke posprema u varijablu (referencu) p

```
package hr.fer.oop.staticmethods;
public class Point
  …
  public static Point centerOf(Point[] points){
            double x = 0, y = 0;
            int len = points.length;
            for(Point p : points){
                  x += p.x;           y += p.y;
            }
            Point p = new Point(x / len, y / len);
            return p;
      }  ...
```

03_Constructors/…/hr/fer/oop/staticmethods/Point.java

Težište polja točaka (3)

- Prije poziva potrebno kreirati i popuniti polje
- Što radi *new Point[] {a, b, c, d}*?
  - stvara polje od 4 elemenata, gdje je svaki element referenca na postojeću točku
    - to su (redom) vrijednosti zapisane u *a, b, c* i *d*

```
package hr.fer.oop.staticmethods;
public class Main {
	public static void main(String[] args) {
		...
		Point d = new Point(7, 3);
		Point[] points = new Point[] {a, b, c, d};
		center = Point.centerOf(points);
		center.print();
		...
```

03_Constructors/…/hr/fer/oop/staticmethods/Main.java

Težište više točaka (1)

- Prethodno rješenje radi s poljem bilo koje veličine, ali nepraktičnost leži u činjenici da se prije poziva mora kreirati polje i popuniti referencama
  - praktičnije bi bilo pozvati metodu kao u donjem primjeru

```
package hr.fer.oop.staticmethods;
public class Main {
  public static void main(String[] args) {
      Point a = new Point(0,0);
      ...
      Point.centerOf(a, b).print();
      Point.centerOf(a, b, c).print();
      Point.centerOf(a, b, c, d).print();
      Point.centerOf(a, b, c, d, new Point(4,8)).print();
            ...
```

03_Constructors/…/hr/fer/oop/staticmethods/Main.java

Težište više točaka (2)

- Metoda može imati varijabilni broj argumenata tako da je zadnje navedeni argument metode oblika *Tip… nazivArgumenta*
  - interno se pretvara u polje tog tipa

```
public class Point
  …
  public static Point centerOf(Point a, Point b, Point...points)
  {
       double x = a.x + b.x;
       double y = a.y + b.y;
       for(Point p : points){
             x += p.x;            y += p.y;
       }
       int len = points.length + 2;
       Point p = new Point(x / len, y / len);
       return p;
  } ...
```

03_Constructors/…/hr/fer/oop/staticmethods/Point.java

Težište više točaka (3)

- Što se događa ako postoji više mogućnosti, npr.

  public static Point centerOf(Point a, Point b, Point... points)

  public static Point centerOf(Point a, Point b, Point c)

  - prevodilac će (ako je moguće) odabrati najspecifičniju metodu gdje je to moguće (moguće je u ovom primjeru)

```
package hr.fer.oop.staticmethods;
public class Main {

  public static void main(String[] args) {
      ...
      Point.centerOf(a, b).print();
      Point.centerOf(a, b, c).print();
      Point.centerOf(a, b, c, d).print();
      ...
```

03_Constructors/…/hr/fer/oop/staticmethods/Main.java

Korištenje klase *Point* u nekoj drugoj klasi

- Vektor u $\mathbb{R}^2$ se može definirati pomoću ishodišta i točke
  - točka (tipa *Point*) koja definira vektor i spremljena je u klasi *Vector*
    može se stvoriti temeljem 2 double broja ili postojeće točke

```
package hr.fer.oop.staticmethods;
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = new Point(p);
	}
	public Vector(double x, double y){
		this.p = new Point(x, y);
	}
	public void print() {
		p.print();
	}
}
```

03_Constructors/…/hr/fer/oop/staticmethods/Vector.java

Referenca ili novi objekt (1)?

[*Vrag* | *Stvar* | *Ljepota* | …] *je u detaljima*

- Što se događa ako se konstruktor promijeni…

```
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = new Point(p);
	}
```

… tako da čuva referencu, umjesto kreiranja novog objekta?

```
public class Vector {
	private Point p;
	public Vector(Point p){
		this.p = p;
	}
```

Referenca ili novi objekt (2)?

- Što bi bio ispis sljedećeg programskog odsječka ovisno o kodu na prethodnom slajdu?

```
Point d = new Point(7, 3);
Vector v = new Vector(d);
v.print();
d.setX(17);
d.setY(13);
v.print();
```

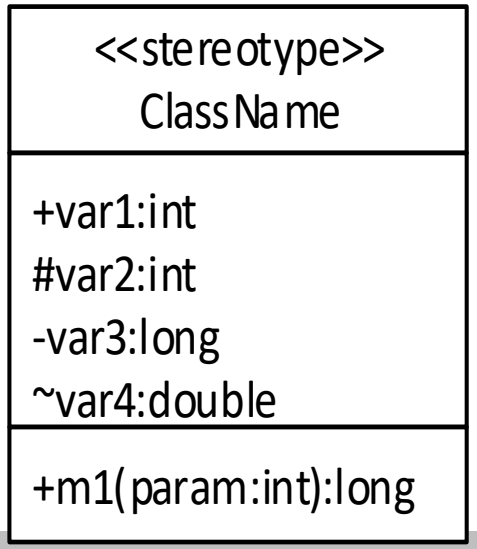
03_Constructors/…/hr/fer/oop/staticmethods/Main.java

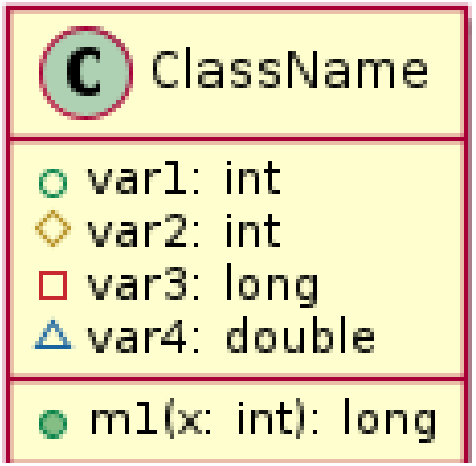
- Odgovor na pitanje iz naslova ovisi o konkretnoj primjeni
  - u ovom slučaju novi objekt (tj. kopija) je željeno ponašanje
  - … ali postoje suprotni slučajevi (npr. liste i općenito kolekcije čuvaju reference)

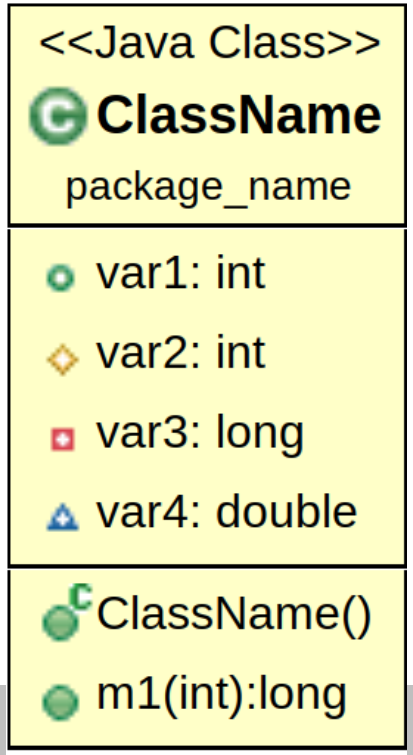
UML dijagram klasa

- UML = Unified Modeling Language
  - grafički jezik za vizualizaciju, specificiranje, konstrukciju i dokumentiranje objektno orijentiranih programskih rješenja
  - neovisan o programskom jeziku
- Dijagram klasa je jedan od UML-ovih dijagrama koji opisuje klase u sustavu te veze između njih
  - prikazuje metode i atribute klasa
- Izgleda može varirati ovisno o korištenom alatu

+ public
 # protected
- private
~ package-private







Primjer alata za prikaz dijagrama klasa

- PlantUML: https://plantuml.com s pluginom za
  - Eclipse: https://marketplace.eclipse.org/content/plantuml-plugin
    - Show View → Other → Plant UML Project Class Diagram i odabrati klase za dijagram
  - NetBeans: https://plugins.netbeans.apache.org/catalogue/?id=58
- Web sučelje PlantUML-a

http://www.plantuml.com/plantuml/uml

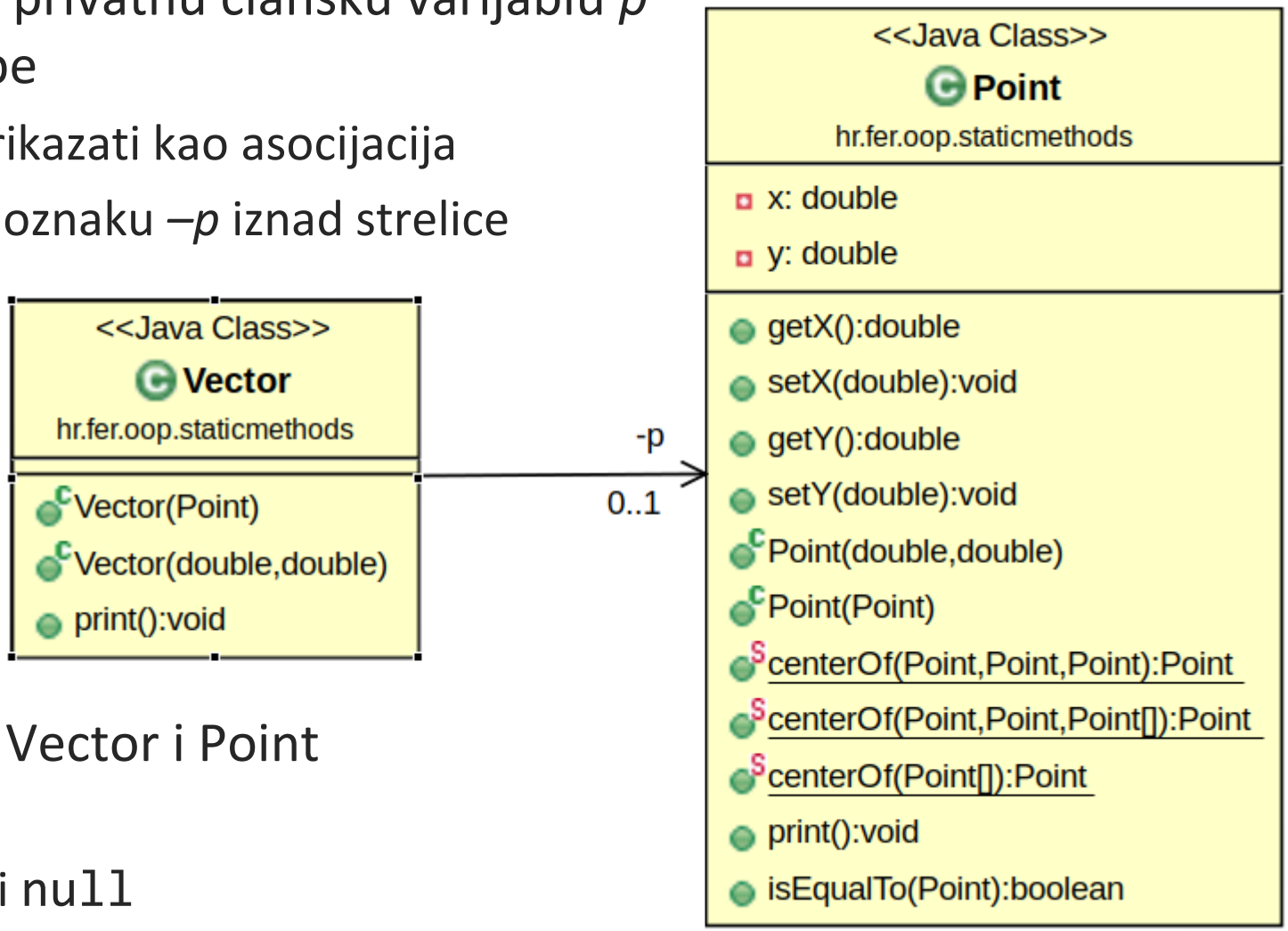
omogućava prikaz klasa iz posebno oblikovanog teksta

```
@startuml
class "<b>ClassName</b>\n<size:8>package_name"
{
        +var1: int
        #var2: int
        -var3: long
        ~var4: double
       m1(x: int): long
}
@enduml
```

- *Napomena: Neki od dijagrama klasa koji će biti prikazani u predavanjima izrađeni su alatom ObjectAid čiji je razvoj napušten, a stare verzije ne rade s novijih verzijama Eclipsea, ali su ostavljeni zbog praktičnosti prikaza*

UML dijagram klasa na primjeru *Vector* i *Point*

- Vector sadrži privatnu člansku varijablu *p* tipa Point type
  - može se prikazati kao asocijacija
  - primijeniti oznaku *–p* iznad strelice



- Veza između Vector i Point is “0 ili 1”
  - *p* može biti `null` ili referenca na neku postojeću točku

Statičke varijable

- Pripada klasi, a ne pojedinom objektu
  - dostupna čak i ako ne postoji niti jedan primjerak te klase
- Koristi se u obliku `NazivKlase.nazivVarijable`
  - Java dopušta da se koristi i nad konkretnim objektom te klase, npr. `obj.nazivVarijable`, ali nije preporučljivo

- Neki primjeri statičkih varijabli:
  - `Math.PI`, `Math.E`
  - obično služe za definiranje konstanti, ali ne mora nužno biti

Ključna riječ *final*

- Vrijednost varijable označena s *final* se ne može mijenjati

```
final int x = 7;
...
x = 5;
```

```
final Point p = new Point(2.5, 3.0);
...
p = new Point(7.0, 4.2);
```

- …ali može se promijeniti objekt na kojeg referenca pokazuje!

```
final Point p = new Point(2.5, 3.0);
...
p.setX(7.0); p.setY(3.0)
```

- mogu biti i statičke
- inicijalizacija odmah pri deklaraciji ili u konstruktoru
  - posljedično, varijabla može biti konstantna na razini pojedinog objekta, a ne na razini klase
- Napomena: ključna riječ *final* se još može koristiti za onemogućavanje nasljeđivanja neke klase i onemogućavanje nadjačavanja metoda (više o tome u 4. i 5. predavanjima)

Primjer: Statičke varijable za bazu vektorskog prostora $\mathbb{R}^2$

- Kanonsku bazu za $\mathbb{R}^2$ čine vektori e_1=(1,0) i e_2=(0,1).
- Bazu mogu činiti i neki drugi vektori, npr. α_1=(1,1) i α_2=(-1, 2)
- Svaki vektor u $\mathbb{R}^2$ se može predstaviti kao linearna kombinacija vektora baze.

- Želimo da je baza ista za sve objekte tipa *Vector?*
  - označit ćemo varijable sa *static*
- Želimo da se kanonska baza ne može promijeniti => *final static*
  - Napomena: označavanje e1 i e2 s *final* znači da su reference konstantne. Budući da u klasi *Vector* nema metoda koje bi mogle promijeniti vektor, niti postoji metoda koja bi vratila referencu na učahurenu točku, onda je u ovom primjeru kanonska baza zbilja nepromjenjiva.

Inicijalizacija statičkih varijabli

- Statičke varijable mogu se inicijalizirati pridruživanjem vrijednosti odmah pri deklaraciji i/ili pisanjem koda unutar bloka *static*
- Prvo se uzimaju (redom) vrijednosti postavljanje pri deklaraciji, a zatim se izvršava blok static.
  - blok *static* će se izvesti prije prvog korištenja statičke varijable ili prije prvog stvaranja objekta te klase.

```
package hr.fer.oop.staticblocks;
public class Vector {
   public final static Vector e1 = new Vector(new Point(1,0));
   public final static Vector e2 = new Vector(new Point(0,1));
   public static Vector alpha1, alpha2;
   static {
            alpha1 = e1; alpha2 = e2;
      } ...
```

03_Constructors/…/hr/fer/oop/staticblocks/Vector.java

Primjer korištenja statičkih varijabli

- Metoda *print* koristi *EquationSolver* za pronalazak linearne kombinacije trenutne baze (ne ulaziti u detalje implementacije!)

03_Constructors/…/hr/fer/oop/staticblocks/Vector.java

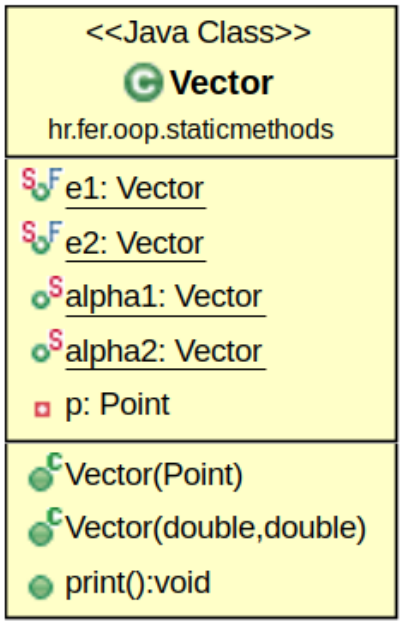
```
package hr.fer.oop.staticblocks;
public class Vector {
  ...
  public void print() {
    System.out.format("(%.2f, %.2f) = %.2f * (%.2f, %.2f) + %.2f
* (%.2f, %.2f)", ... //details are not relevant!
```

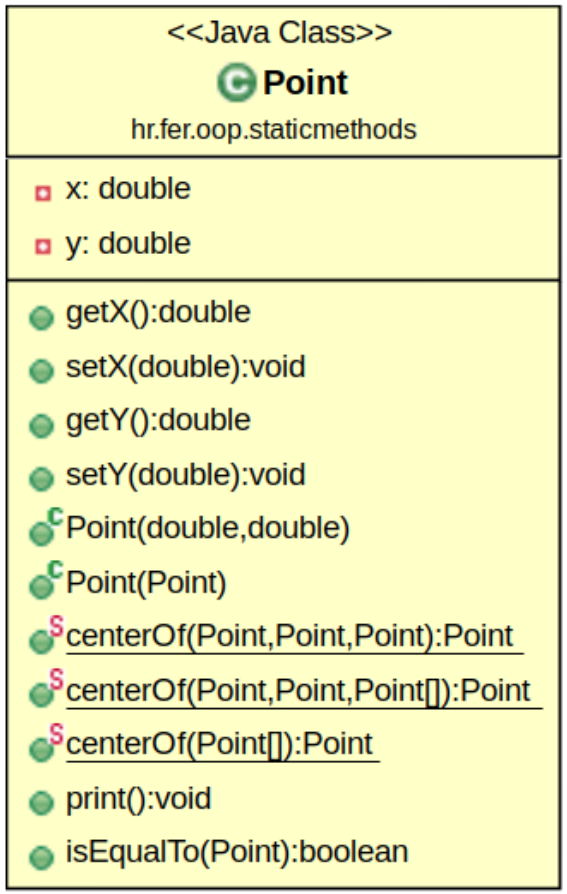
03_Constructors/…/hr/fer/oop/staticblocks/Main.java

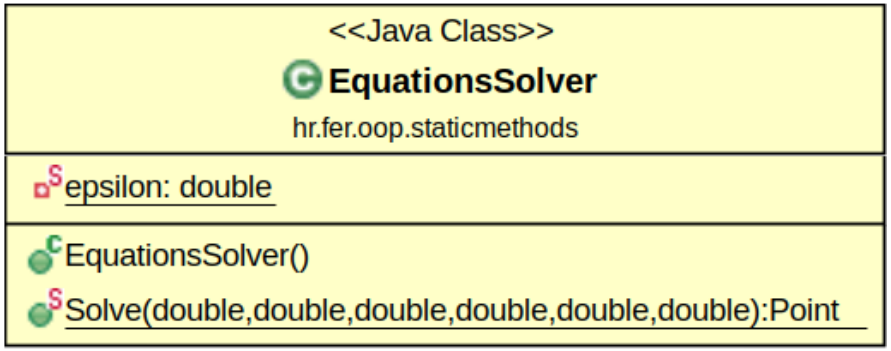
```
public class Main {
  Vector v = new Vector(new Point(3,4));
  v.print();
  Vector.alpha1 = new Vector(1,1);
  Vector.alpha2 = new Vector(-1,2);
  v.print();
```

```
(3,00, 4,00) = 3,00 * (1,00, 0,00) + 4,00 * (0,00, 1,00)
(3,00, 4,00) = 3,33 * (1,00, 1,00) + 0,33 * (-1,00, 2,00)
```

Dijagram klasa nakon promjena u klasama *Vector* i *Point*







- Veze između klasa nisu prikazane radi preglednosti

Dijagram klasa prikazan alatom PlantUML