

# Objektno orijentisano programiranje 2

Izvođenje



# Proširivanje klasa

- Ugovor klase:
  - kolekcija spolja pristupačnih metoda (i eventualno polja), zajedno sa semantikom
- Proširiti klasu znači proširiti ugovor
- Nasleđeni deo ugovora ne treba da se menja
  - objekat izvedene klase treba da zamenjuje objekat osnovne
- Razumna je promena implementacije ugovora natklase
  - promena implementacije znači redefinisavanje metoda

# Primer - Stek

- Klasa `Stek` koja se proširuje da bi se dobila klasa steka sa prioritetima:

```
class Stek {
    protected int sp=0;
    protected Object[] niz;
    public Stek(int velicina){
        niz=new Object[velicina];
    }
    public void resetuj(){
        sp=0; for(int i=0;i<niz.length;i++)niz[i]=null;
    }
    public void stavi(Object o){
        niz[sp++]=o;
    }
    public Object uzmi(){
        Object o=niz[--sp]; niz[sp]=null; return o;
    }
}
```

- Napomena: klasa `Stek` ne rešava probleme prekoračenja i uzimanja sa praznog steka

# Primer – PriorStek (1)

```
public class PriorStek extends Stek{
    private static final int MIN_PRIOR=0;
    private int[] prioriteti;
    public PriorStek(int velicina){
        super(velicina);
        prioriteti=new int[velicina];
    }
    public void stavi(Object o){
        super.stavi(o);
        prioriteti[sp-1]=MIN_PRIOR;
    }
    public void stavi(Object o, int p){
        super.stavi(o);
        prioriteti[sp-1]=p;
    }
    //...
```

## Primer - PriorStek (2)

```
public Object uzmi() {
    int maxPri=maxPrioritet();
    Object o1,o2;
    int p1,p2,spp=--sp;
    o1=niz[spp];
    niz[spp]=null;
    p1=prioriteti[spp];
    prioriteti[spp]=MIN_PRIOR;
    while (p1!=maxPri && --spp>=0) {
        o2=niz[spp];
        p2=prioriteti[spp];
        niz[spp]=o1;
        prioriteti[spp]=p1;
        o1=o2; p1=p2;
    }
    return o1;
}

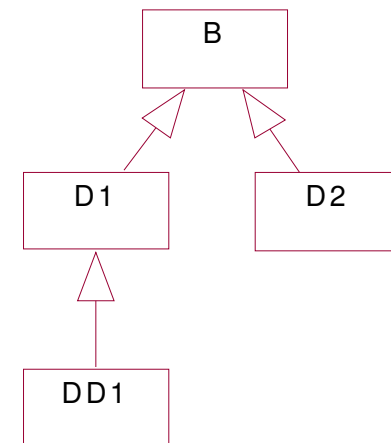
private int maxPrioritet(){
    int maxPri=MIN_PRIOR;
    for(int i=0; i<sp; i++)
        if (prioriteti[i]>maxPri)
            maxPri=prioriteti[i];
    return maxPri;
}
```

# Korena klasa Object

- Koren cele hijerarhije klasa predstavlja klasa `Object`
- Klase koje ne koriste `extends` klauzulu
  - implicitno proširuju klasu `Object`
- Promenljive (reference) tipa `Object`
  - mogu da upućuju na proizvoljan objekat
  - ne mogu da upućuju na primitivne tipove
    - ali mogu da upućuju na objekte klasa omotača (*wrapping classes*), npr. `Integer`, `Boolean`
  - od Java 5.0 postoji “samopakovanje” (*autoboxing*)
    - ispravno je čak pisati: `Object o=100;`
    - automatski se stvara objekat klase `Integer`
    - argument konstruktora je vrednost `100` tipa `int`
    - referenca `o` može da pokazuje na objekat tipa `Integer`

# Značenje protected

- Zaštićenom članu se može pristupiti iz izvedene klase
  - direktnim imenovanjem
  - preko referenci na objekte koje su tipa:
    - te izvedene klase
    - neke klase izvedene iz te izvedene klase
- Primer:
  - neka je u klasi B: `protected int x;`
  - kod u klasi D1 može da pristupi x
    - direktno, t.j. preko reference `this`
    - preko referenci tipa D1 ili DD1
  - kod u klasi D1 ne može da pristupi x
    - preko referenci tipa D2 ili B
- Zaštićenim statičkim članovima se može pristupiti preko osnovne B i bilo koje izvedene klase (npr. preko D2)
- Zaštićeni članovi su pristupačni celom kodu unutar paketa



# Konstruktori

- Izvedena klasa mora da pozove jedan od konstruktora svoje osnovne klase da obezbedi korektno inicijalno stanje nasleđenih polja
- Eksplicitan poziv konstruktoru osnovne klase (“superklase”):  
`super(<lista argumenata>)`
  - ako se eksplicitno koristi `super()`
    - to mora biti prva naredba u novom konstruktoru
  - ako se ne koristi – *no-arg* konstruktor osnovne klase će biti pozvan implicitno
  - ako osnovna klasa nema *no-arg* konstruktor
    - obavezan eksplicitan poziv konstruktora osnovne klase
- `this(<lista argumenata>)` poziva odgovarajući vlastiti konstruktor
- Ako postoji, `this(<lista argumenata>)`
  - mora biti prva naredba u konstruktoru
  - tada ne može postojati i `super(...)`
  - praktično, odluka koji će se konstruktor osnovne klase pozvati se može zaobići pozivom drugog konstruktora iste klase `this(...)`



# Podrazumevani konstruktor

- Podrazumevani konstruktor je
  - bez argumenata (*no-arg*)
  - sa telom koje ima samo poziv `super()`
- Podrazumevani konstruktor u javnoj izvedenoj klasi `A` je ekvivalentan

```
public class A extends B{
    public A(){ super(); }
}
```
- Konstruktor je javni, ako je klasa javna
  - npr. `A` je javna, pa je i konstruktor
- Jezik ga obezbeđuje automatski, ukoliko nije napisan neki konstruktor klase
- Ako osnovna klasa nema *no-arg* konstruktor, izvedena klasa mora definisati konstruktor

# Redosled konstrukcije objekata

- Kada se kreira objekat, njegovi atributi (polja) se postave na podrazumevane vrednosti, prema tipu atributa:
  - nula (numerički), `\u0000` (char), `false` (boolean), `null` (referenca)
- Konstruktor ima 3 faze:
  - (1) poziv konstruktora superklase
  - (2) inicijalizacija polja koristeći njihove inicijalizatore i nestatičke inicijalizacione blokove sleva-udesno, odozgo-naniže
  - (3) izvršavanje tela konstruktora
- Ako se pozove metod za vreme konstrukcije
  - izvršiće se za objekat u tekućem stanju
  - neka polja objekta mogu još uvek biti neinicijalizovana
  - takve metode treba projektovani pažljivo

# Primer redosleda

```
class X{
    protected int xMaska = 0x00ff;
    protected int celaMaska;
    public X(){celaMaska = xMaska;}
    public int maska(int orig)
        {return(orig & celaMaska);}
}

class Y extends X{
    protected int yMaska = 0xff00;
    public Y(){celaMaska |= yMaska;}
}

Y y=new Y();
```

	Akcija	xMaska	yMaska	celaMaska
0	Polja ←podraz. vred.	0	0	0
1	Y() pokrenut	0	0	0
2	X() pokrenut	0	0	0
3	X inicijalizacija polja	00ff	0	0
4	X() izvršen	00ff	0	00ff
5	Y inicijalizacija polja	00ff	ff00	00ff
6	Y() izvršen	00ff	ff00	ffff

- Ako bi konstruktor X() pozvao maska(orig) vršilo bi se maskiranje sa 0x00ff, a ne sa 0xffff

# Redefinisanje (nadjačanje) metoda

- Preklapanje imena (*overloading*)
  - više metoda ima isto ime
  - obavezna je razlika u potpisima metoda
  - mehanizam povezivanja je statički
- Redefinisanje (nadjačanje) metoda (*overriding*)
  - zamena implementacije metoda iz osnovne klase
  - potpisi i povratni tipovi moraju biti identični (uz izuzetak kao u C++)
  - mehanizam povezivanja je dinamički
- Samo nestatički i ne privatni metodi se mogu polimorfno redefinisati
- Referenca `super` se može koristiti da pristupi originalnim metodima osnovne klase
- Klauzula `throws` može biti različita od `throws` klauzule datog metoda osnovne klase
  - metod izvedene klase može da suzi listu `throws` klauzule

# Pravo pristupa redefinisanim metodima

- Redefinisani metodi imaju svoje vlastite modifikatore
- Izvedena klasa može promeniti pravo pristupa metodima natklase dajući im širi pristup
  - `protected` metod može biti redefinisani kao `protected` ili `public`, ali ne i `private`
- Smanjivanje prava pristupa nije dozvoljeno i ne bi imalo smisla:
  - restrikcija bi se zaobišla konverzijom tipa u osnovni tip sa širim pravom pristupa

# Sakrivanje polja i polimorfizam

- Polja ne mogu da budu redefinisana, ali mogu da budu sakrivena
- Ako se u izvedenoj klasi deklarise polje sa istim imenom kao u osnovnoj klasi  $\Rightarrow$  2 polja
- Referenca na osnovnu klasu (`super`) se mora koristiti da se pristupi polju osnovne klase
- Metodi u Javi su podrazumevano polimorfni
- Kada se pozove metod objekta
  - stvarni tip objekta (ne reference) određuje koja se implementacija metoda koristi (polimorfno ponašanje)
- Kada se pristupa polju
  - koristi se tip reference (nepolimorfno ponašanje)

# Primer

```
class B{
    public String s="B";
    public void p(){ System.out.println("B.p(): " + s); }
}
class D extends B{
    public String s = "D";
    public void p(){ System.out.println("D.p(): " + s); }
    public static void main(String[] arg){
        D d = new D();
        B b = d;
        b.p();
        d.p();
        System.out.println("B.s: " + b.s);
        System.out.println("D.s: " + d.s);
    }
}
```

## Izlaz:

```
D.p(): D
D.p(): D
B.s: B
D.s: D
```

# Referenca super

- Na raspolaganju u nestatičkim metodima izvedene klase
- `super` se ponaša kao
  - referenca na tekući objekat koji je primerak svoje osnovne klase
- Poziv `super.metoda` koristi implementaciju metoda osnovne klase
- Poziv metoda preko `super` je različit od drugih poziva preko referenci:
  - izbor je zasnovan na tipu reference a ne objekta (ne primenjuje se dinamičko vezivanje)
- Nije dozvoljeno:
  - `B b=super;`
  - ... `super.super.m()` ...



# Primer

```
class B{ protected String m(){ return "B"; } }
class D extends B{
    protected String m(){ return "D"; }
    void p(){
        B bref = (B)this; // (B)this semantički odgovara super;
        System.out.println("this.m() = " + this.m());
        System.out.println("bref.m() = " + bref.m());
        System.out.println("super.m() = " + super.m());
    }
}
```

## **Izlaz:**

```
this.m() = D (jer this upućuje na tekući objekat)
bref.m() = D (jer i bref upućuje na tekući objekat)
super.m() = B (jer pristup preko super nije polimorfan)
```

# Kompatibilnost tipova

- Java je strogo tipiziran jezik
  - intenzivno se proverava kompatibilnost tipova prilikom prevođenja
- Tip izraza mora biti kompatibilan tipu promenljive kojoj se rezultat izraza dodeljuje i to pri:
  - operaciji dodele,
  - inicijalizaciji,
  - prenosu argumenata i
  - vraćanju rezultata metoda
- Tip reference koja je rezultat izraza mora biti:
  - tip promenljive kojoj se dodeljuje
  - izvedeni tip iz tipa promenljive kojoj se dodeljuje
  - literal `null` (kompatibilan sa svim tipovima referenci)

# Konverzija

- Nadtipovi su manje posebni – oni su "širi" i nalaze se "više" u hijerarhiji
- Podtipovi su više posebni – oni su "uži" i nalaze se "niže" u hijerarhiji
- Konverzija proširivanja (naviše, nagore):
  - podtip se konvertuje u nadtip
  - bezbedna konverzija – automatski
- Konverzija sužavanja (naniže, nadole):
  - nadtip se konvertuje u podtip
  - nije bezbedna, treba biti oprezan – mora eksplicitno (*cast*)
- Primer: (u metodu izvedene klase D iz klase B)

```
B b = (B) this; // može i: B b = this;  
D d = (D) b; // obavezan cast
```
- Ako prevodilac može da zaključi da je konverzija sužavanja neispravna
  - javlja grešku
- Ako prevodilac ne može sa sigurnošću da zaključi da je konverzija pogrešna
  - radi se provera u vreme izvršenja i u slučaju greške – `ClassCastException`

# Operator provere tipa `instanceof`

- Operator `instanceof` služi za proveru da li je izraz kompatibilan za dodelu datom tipu reference
- Izraz je levi operand (rezultuje u referenci na objekat), a ime (nep primitivnog) tipa je desni operand
- Provera za dodelu reference osnovne klase `B` `b` referenci izvedene kl. `D` `d`:

```
if (b instanceof D) d=(D)b;
```
- Primer:

```
public static void sortiraj (Lista l){  
    if (l instanceof SortiranaLista) return;  
    // sortiranje liste  
}
```
- Ovaj operator ne treba "zloupotrebljavati"
  - ne treba ga koristiti kao zamenu za redefinisane i polimorfno korišćenje metoda
  - nije dobro da se:
    - ispituje tip izraza pomoću `instanceof` u nekoj selekciji (`if/switch`)
    - pa na osnovu rezultata testova, u pojedinim granama pozivati različite metode objekta na koji ukazuje referenca koja je rezultat izraza levog operanda (`b`)

# Konačni metodi i klase

- Označavanje metoda kao `final` ne dozvoljava izvedenim klasama da ga redefinišu
- Cela klasa može biti označena kao konačna (finalna):  

```
final class F{//...}
```
- Konačna klasa se ne može proširivati (te su joj i svi metodi automatski konačni)
- Razlozi da se metod proglasi konačnim:
  - sigurnost:  
ponašanje klase se neće promeniti, bez obzira na stvarni tip objekta
  - optimizacija:  
nema dinamičkog vezivanja, provere tipova se mogu vršiti u vreme prevođenja
- U pogledu optimizacije
  - `final` metodi su ekvivalentni sa `private` i `static` metodima
  - na privatne i statičke metode se ne primenjuje dinamičko vezivanje

# Apstraktni metodi i klase

- Apstrakcija je korisna kada neko ponašanje ima smisla samo za pojedine podtipove
- Apstraktan metod (modifikator `abstract`)
  - metod čija će implementacija biti definisana tek u izvedenoj klasi
- Apstraktna klasa (modifikator `abstract`)
  - način da se definiše tip sa delimičnom implementacijom
- Klasa sa barem jednim apstraktnim metodom mora biti apstraktna
- I klasa sa svim konkretnim metodima može da bude apstraktna
- Ako je potrebno da svi metodi budu apstraktni, verovatno je potreban interfejs (ne klasa)
- Ne mogu se kreirati objekti apstraktne klase
  - reference na apstraktnu klasu su dozvoljene
- Običan metod može biti redefinisana kao apstraktan

# Primer

```
abstract class Testiranje {
    abstract void test();
    public long izvrsi(int n){
        long start = System.currentTimeMillis();
        for (int i = 0; i < n; i++) test();
        return (System.currentTimeMillis() - start);
    }
}
class TestiranjePraznogMetoda extends Testiranje {
    void test() {}
    public static void main(String[] argumenti) {
        int n = Integer.parseInt(argumenti[0]);
        long vreme = new TestiranjePraznogMetoda().izvrsi(n);
        System.out.println(n+" poziva metoda za "+vreme+" ms");
    }
}
```

# Klasa Object

- Sve klase se izvode (implicitno ili indirektno) iz klase `Object`
- Klase nasleđuju sledeće metode klase `Object` :
  - poređenje objekata na jednakost (`==` se koristi za proveru jednakosti referenci)

```
public boolean equals(Object obj)
```

    - podrazumevano proverava da li je `obj==this`  
(podrazumevano samo `o.equals(o)==true`)
  - kloniranje objekta (novi objekat preuzima stanje onog koji se klonira)

```
protected Object clone() throws CloneNotSupportedException
```
  - dohvatanje `Class` objekta koji opisuje tip (klasu, interfejs, niz) datog objekta

```
public final Class getClass()
```
  - finalizacija objekta za vreme sakupljanja đubreta

```
protected void finalize() throws Throwable
```
  - konverzija u `String`

```
public String toString()
```

    - objekat se konvertuje u string `imeKlase@<heksadecimalni heš kod>`



# Kloniranje objekata

- Metod `clone()` vraća novi objekat čije je inicijalno stanje kopija tekućeg stanja objekta
- Promene klona više ne utiču na original
- Klasa koja omogućava kloniranje
  - treba da implementira (prazan) interfejs `Cloneable`
- Metod `Object.clone()`
  - obavlja jednostavno (plitko) kloniranje kopiranjem svih polja originala
- Izuzetak `CloneNotSupportedException`
  - koristi se da signalizira da `clone()` metod nije uspeo
- Kloniranjem se izbegava pozivanje konstruktora

# Kreiranje klona

- Koraci `Object.clone()`:
  - proverava da li objekat za koji je metod pozvan implementira interfejs `Cloneable`
  - ako ne implementira, baca izuzetak `CloneNotSupportedException`
  - inače, stvara novi objekat istog tipa kao i originalni objekat i kopira polja
  - vraća referencu tipa `Object` na novi klon
- Najjednostavniji način da se napravi klasa koja (uslovno) omogućava kloniranje, ali samo plitko:

```
public class MojaKlasa extends DrugaKlasa implements Cloneable {
    public MojaKlasa clone() throws CloneNotSupportedException{
        return (MojaKlasa) super.clone();
    }
}
```

  - sva polja `MojaKlasa` ce biti kopirana pomoću `Object.clone()`
  - uslov: da `DrugaKlasa` i njeni preci podržavaju kloniranje (realizuju `clone()` koji poziva `super.clone()`)
- Ako klasa ima polja tipa reference, po pravilu je potreban duboki klon

# Primer – plitki klon

- Primer neodgovarajućeg plitkog kloniranja:

```
public class CelobrojniStek implements Cloneable{
    private int[] bafer; private int vrh;
    public CelobrojniStek(int kapacitet)
        { bafer = new int[kapacitet];    vrh = -1; }
    public void stavi(int v){ bafer[++vrh] = v; }
    public int uzmi(){ return bafer[vrh--]; }
    public CelobrojniStek clone() throws CloneNotSupportedException
        { return (CelobrojniStek) super.clone(); }
}
```

...

```
CelobrojniStek prvi = new CelobrojniStek(2);
prvi.stavi(1); prvi.stavi(2);
try{CelobrojniStek drugi = prvi.clone()}
catch(CloneNotSupportedException i){};
```

- **obe reference** prvi.bafer i drugi.bafer ukazuju na istu lokaciju

# Primer – duboki klon

- Rešenje problema – duboko kloniranje:

```
// redefinisavanje clone() metoda da se postigne kopiranje niza
public CelobrojniStek clone(){
    try {
        CelobrojniStek noviStek = (CelobrojniStek)super.clone();
        noviStek.bafer = (int[])bafer.clone();
        return noviStek;
    } catch(CloneNotSupportedException i) { // nece se dogoditi
        throw new InternalError(i.toString());
    }
}
```

# Naklonosti klase prema klonovima

- 4 nivoa naklonosti prema klonovima:
  - klasa i potklase bezuslovno podržavaju kloniranje:
    - klasa implementira `Cloneable` i
    - deklarira svoj javni `clone()` metod koji ne baca izuzetke
  - klasa uslovno podržava kloniranje:
    - implementira `Cloneable`,
    - dopušta svom javnom `clone()` metodu da prosledi izuzetak dobijen od drugih objekata čije je kloniranje pokušao
  - klasa ne podržava kloniranje, ali dopušta potklasi da podrži kloniranje:
    - klasa ne implementira `Cloneable`,
    - ali obezbeđuje zaštićen `clone()` metod koji klonira njena polja korektno (ako `Object.clone()` ne zadovoljava)
  - klasa sprečava kloniranje svojih objekata i objekata potklasa:
    - klasa ne implementira `Cloneable` i
    - obezbeđuje javni konačni `clone()` metod koji bezuslovno baca izuzetak

# Izvođenje ili sadržanje: kako i kada?

- Izvođenje deviniše novi tip objekata koji su i tipa osnovne klase
  - *IsA* relacija
- Relacija *IsA* je potpuno različita od *HasA* relacije
- U relaciji *HasA* jedan objekat
  - čuva stanje drugog (sadrži drugi)
  - sadržani često radi neki deo posla sadržaoaca
- Primeri:
  - *Piksel IsA Tačka* sa proširenjem (informacija o boji)
  - *Krug HasA Tačka* (kao centar)
- Ponekad dobar izbor između dve relacije nije očigledan
- Promena lošeg inicijalnog izbora može biti teška

# Primer problematičnog izvođenja (1)

- Klasa tačka u 2D

```
class Tacka2D{
    protected double x=0., y=0.;
    Tacka2D() {}
    Tacka2D(double x, double y){ this.x=x; this.y=y; }
    Tacka2D(Tacka2D p){ this(p.x,p.y); }
    public double rastojanje(Tacka2D p){
        double dx=x-p.x, dy=y-p.y;
        return Math.sqrt(dx*dx+dy*dy);
    }
    public String toString(){ return "("+x+", "+y+")"; }
}
```

# Primer problematičnog izvođenja (2)

- Klasa Tacka2D se proširuje poljem z da bi se dobila tačka u 3D

```
class Tacka3D extends Tacka2D{
    private double z=0.;
    Tacka3D(){}
    Tacka3D(double x, double y, double z){
        super(x,y); this.z=z;
    }
    Tacka3D(Tacka3D p){ this(p.x,p.y,p.z); }
    public double rastojanje(Tacka3D p){
        double d2D=super.rastojanje(p), dz=z-p.z;
        return Math.sqrt(d2D*d2D+dz*dz);
    }
    public String toString(){ return "("+x+", "+y+", "+z+")"; }
}
```

- Metod rastojanje() nije nadjačanje, nego preklapanje imena