

# Programiranje 1

## Poglavje 7: Dedovanje

Luka Fürst

# Dedovanje



- relacija med razredi (tipi)
- tip  $B$  je poseben primer tipa  $A$
- objekti tipa  $B$  imajo vse lastnosti, ki jih imajo objekti tipa  $A$ , lahko pa imajo še dodatne

# Vsakdanji primeri

- izredni študent  $\leftarrow$  študent
  - vsak izredni študent je tudi študent
  - izredni študent je poseben primer študenta
  - izredni študent ima vse lastnosti študenta, lahko pa ima še dodatne
- pes  $\leftarrow$  sesalec  $\leftarrow$  vretenčar  $\leftarrow$  žival
- vrstna hiša  $\leftarrow$  hiša  $\leftarrow$  nepremičnina
- kvadrat  $\leftarrow$  pravokotnik  $\leftarrow$  štirikotnik  $\leftarrow$  lik

## Sintaksa in terminologija

```
class A {  
    ...  
}  
  
class B extends A {  
    ...  
}
```

- »razred B je **podrazred** razreda A«
- »razred A je **nadrazred** razreda B«
- »razred B je **izpeljan** iz razreda A«
- »razred B **razširja** razred A«
- »tip B je **podtip** tipa A«
- »tip A je **nadtip** tipa B«

# Dedovanje

```
class A { ... }  
  
class B extends A { ... }
```

- razred B **podeduje** od razreda A
  - vse attribute (tudi privatne)
  - vse metode razen privatnih
- konstruktorji se ne dedujejo!
- razred B lahko doda svoje attribute in metode
- razred B lahko posamezne metode razreda A **redefinira**

# Razred Student

- atributi
  - `private String ip`: ime in priimek
  - `private String vpisna`: vpisna številka
  - `private int stroskiBivanja`: mesečni stroški bivanja
- konstruktor
  - `public Student(String ip, String vpisna, int stroskiBivanja)`: nastavi vse tri attribute
- metodi
  - `public String vrniIP()`: vrne ime in priimek
  - `public int stroski()`: vrne mesečne stroške študija (ti so kar enaki stroškom bivanja)

# Razred Student

```
public class Student {  
    private String ip;  
    private String vpisna;  
    private int stroskiBivanja;  
  
    public Student(String ip, String vpisna, int stroskiBivanja) {  
        this.ip = ip;  
        this.vpisna = vpisna;  
        this.stroskiBivanja = stroskiBivanja;  
    }  
  
    public String vrniIP() {  
        return this.ip;  
    }  
  
    public int stroski() {  
        return this.stroskiBivanja;  
    }  
}
```

# Razred IzredniStudent

- to se podeduje:
  - vsi atributi (`ip`, `vpisna` in `stroskiBivanja`)
  - obe metodi (`vrniIP` in `stroski`)
- to bomo dodali:
  - atribut `private int solnina` (mesečna šolnina)
  - `konstruktor` (obvezno)
- `redefinirali` bomo metodo `stroski`
  - mesečni stroški študija se za izrednega študenta izračunajo drugače kot za splošnega študenta



## Razred IzredniStudent

```
public class IzredniStudent extends Student {  
    // dodatni atribut  
    private int solnina;  
  
    // konstruktor  
    public IzredniStudent(...) {  
        ...  
    }  
  
    // redefinirana metoda  
    @Override  
    public int stroski() {  
        ...  
    }  
}
```

## Konstruktor razreda IzredniStudent

- nastavi vrednosti vseh štirih atributov
- prvi poskus

```
public IzredniStudent(String ip, String vpisna,  
                        int stroskiBivanja, int solnina) {  
    this.ip = ip;  
    this.vpisna = vpisna;  
    this.stroskiBivanja = stroskiBivanja;  
    this.solnina = solnina;  
}
```

- gornja koda se ne prevede!

# Konstruktor razreda IzredniStudent

- problem št. 1
  - atributi ip, vpisna in stroskiBivanja niso neposredno dostopni, saj so v razredu Student deklarirani kot privatni
  - attribute lahko v razredu Student deklariramo kot `protected`, vendar tega brez tehtnih razlogov naj ne bi počeli


# Konstruktor razreda IzredniStudent

- problem št. 2
  - konstruktor podrazreda naj bi se pričel s stavkom `super( $p_1, \dots, p_n$ )`, ki pokliče konstruktor nadrazreda in mu posreduje parametre  $p_1, \dots, p_n$
  - če stavek `super(...)` izpustimo, prevajalnik na začetek konstruktorja sam doda stavek `super()` (brez parametrov)
  - če nadrazred ne vsebuje konstruktorja brez parametrov, potem stavek `super()` povzroči napako
- če nadrazred nima konstruktorja brez parametrov, potem se konstruktor podrazreda mora pričeti s `super(...)`

## Konstruktor razreda IzredniStudent

- najprej pokličemo konstruktor nadrazreda, da nastavi attribute ip, vpisna in stroskiBivanja
- nato nastavimo še atribut solnina

```
public IzredniStudent(String ip, String vpisna,  
                        int stroskiBivanja, int solnina) {  
    super(ip, vpisna, stroskiBivanja);  
    this.solnina = solnina;  
}
```



## Redefinicija metode

```
class A {  
    določilo T metoda( $T_1 p_1, \dots, T_n p_n$ ) { ... }  
}  
class B extends A {  
    @Override  
    določilo' T metoda( $T_1 p_1, \dots, T_n p_n$ ) { ... }  
}
```

- redefinirana metoda mora v podrazredu imeti enako ime, enako zaporedje tipov parametrov in enak izhodni tip kot v nadrazredu
- dostopno določilo mora biti isto ali ohlapnejše
- pred glavo metode v podrazredu dodamo oznako **@Override**
  - ni obvezna, a zmanjša verjetnost napake
  - prevajalnik preveri, ali nadrazred vsebuje metodo z enako glavo


## Redefinicija metode

- znotraj redefinirane metode v podrazredu lahko uporabimo `super.metoda( $d_1, \dots, d_n$ )`
- pokliče različico te metode v nadrazredu in ji posreduje parametre  $d_1, \dots, d_n$
- klic `super.metoda(...)` ni obvezen
- lahko ga uporabimo kjerkoli v metodi (tudi večkrat)

## Metoda stroski

- stroški študija se za izrednega študenta izračunajo tako, da se **splošnim študentovim stroškom** prišteje še **šolnina**

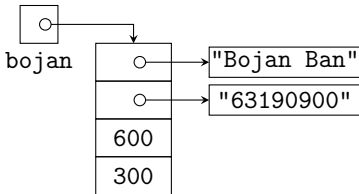
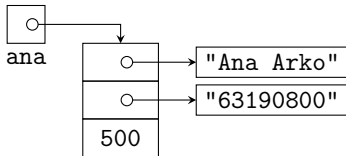
```
public class IzredniStudent extends Student {  
    ...  
    @Override  
    public int stroski() {  
        return (super.stroski() + this.solnina);  
    }  
}
```





# Testni razred

```
public class TestStudent {  
    public static void main(String[] args) {  
        Student ana = new Student("Ana Arko", "63190800", 500);  
        System.out.println(ana.vrniIP());    // Ana Arko  
        System.out.println(ana.stroski());   // 500  
  
        IzredniStudent bojan = new IzredniStudent(  
            "Bojan Ban", "63190900", 600, 300);  
        System.out.println(bojan.vrniIP());  // Bojan Ban  
        System.out.println(bojan.stroski()); // 900  
    }  
}
```



## Posebno pravilo pri prirejanju kazalcev

- recimo, da je razred B podrazred razreda A
- kazalec tipa B lahko priredimo spremenljivki tipa A
- obratno ni mogoče

```
A a = new A(...);  
B b = new B(...);  
A p = b;      // OK  
B q = a;      // napaka pri prevajanju
```

- metodi, ki sprejme parameter tipa A, lahko podamo tudi kazalec tipa B

## Tip kazalca in tip objekta

- razred B je podrazred razreda A

```
A a = new A(...);  
B b = new B(...);  
A p = b;
```

- kazalec p je deklariran kot spremenljivka tipa A
- dejansko pa p kaže na objekt tipa B
- tip kazalca p je A
- tip objekta, na katerega kaže p, je B

# Tip kazalca in tip objekta

- prevajalnik (javac) pozna samo tipe kazalcev
  - vidi samo deklaracije
- izvajalnik (java) pozna samo tipe objektov
  - ve, s katerim konstruktorjem so bili ustvarjeni posamezni objekti

## Izbira metode pri klicu

- Katera metoda stroski se bo poklicala?

```
Student ana = new Student("Ana Arko", "63190800", 500);  
IzredniStudent bojan = new IzredniStudent(  
    "Bojan Ban", "63190900", 600, 300);  
Student s = bojan;  
System.out.println(s.stroski());
```

- izbira metode je določena s tipom objekta, ne kazalca
- ker kazalec s kaže na objekt tipa IzredniStudent, se bo poklicala metoda stroski iz razreda IzredniStudent
- rezultat bo potemtakem 900 in ne 600

## Dedovanje in tabele

- vsi elementi tabele so istega tipa
- vendar: če so elementi kazalci tipa  $R$ , lahko kažejo tudi na objekte poljubnih podtipov tipa  $R$
- tabela tipa `Student[]` lahko torej hrani kazalce na objekte tipa `Student` ali `IzredniStudent`
- če se sprehodimo po tabeli in za vsak kazalec v njej pokličemo metodo `stroski`, se bo
  - za kazalce na objekte tipa `Student` poklicala metoda `stroski` iz razreda `Student`
  - za kazalce na objekte tipa `IzredniStudent` poklicala metoda `stroski` iz razreda `IzredniStudent`

## Dedovanje in tabele

```
Student[] studentje = {  
    new Student("Ana Arko", "63190800", 500),  
    new IzredniStudent("Bojan Ban", "63190900", 600, 300),  
    new IzredniStudent("Cvetka Cevc", "63191000", 400, 350),  
    new Student("Denis Denk", "63191100", 450)  
};  
  
for (Student student: studentje) {  
    System.out.printf("%s: %d%n",  
                        student.vrniIP(), student.stroski());  
}
```

```
Ana Arko: 500      // metoda stroski iz razreda Student  
Bojan Ban: 900     // metoda stroski iz razreda IzredniStudent  
Cvetka Cevc: 750   // metoda stroski iz razreda IzredniStudent  
Denis Denk: 450    // metoda stroski iz razreda Student
```

# Hierarhija likov

- napisati želimo program za obdelavo podatkov o likih — pravokotnikih, kvadratih in krogih
  - izpis podatkov
  - računanje ploščine
  - računanje obsega
- podatke o likih bi radi hranili v tabeli
- like bi radi obravnavali karseda poenoteno



# Hierarhija likov

- problem: liki imajo različne attribute
  - pravokotnik: `sirina` in `visina`
  - kvadrat: `stranica`
  - krog: `polmer`
- rešitev: skupni nadrazred `Lik`
- razrede `Pravokotnik`, `Kvadrat` in `Krog` definiramo kot podrazrede razreda `Lik`
- kvadrat je poseben primer pravokotnika, zato lahko razred `Kvadrat` definiramo kot podrazred razreda `Pravokotnik`

# Hierarhija likov

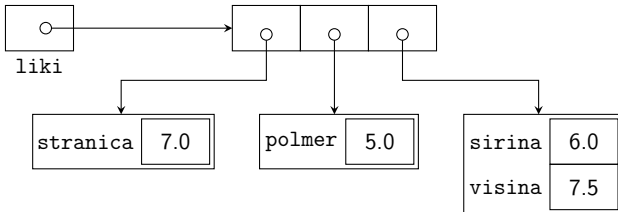
```
public class Lik {  
    ...  
}  
  
public class Pravokotnik extends Lik {  
    ...  
}  
  
public class Kvadrat extends Pravokotnik {  
    ...  
}  
  
public class Krog extends Lik {  
    ...  
}
```

# Hierarhija likov

- hierarhija nam omogoča, da izdelamo tabelo kazalcev na objekte različnih podrazredov razreda Lik



```
Lik[] liki = {  
    new Kvadrat(7.0),  
    new Krog(5.0),  
    new Pravokotnik(6.0, 7.5)  
};
```



# Hierarhija likov

- hierarhijo želimo zasnovati tako, da bomo lahko elemente tabele poenoteno obravnavali

```
for (Lik lik: liki) {  
    System.out.println(lik.toString());  
    System.out.printf("p = %.1f, o = %.1f%n",  
                      lik.ploscina(), lik.obseg());  
    System.out.println();  
}
```

```
kvadrat [stranica = 7.0]
```

```
p = 49.0, o = 28.0
```

```
krog [polmer = 5.0]
```

```
p = 78.5, o = 31.4
```

```
pravokotnik [širina = 6.0, višina = 7.5]
```

```
p = 45.0, o = 27.0
```

## Razred Lik

- če želimo, da se sledeča koda prevede ...

```
for (Lik lik: liki) {  
    System.out.println(lik.ploscina());  
    System.out.println(lik.obseg());  
}
```

- ... potem mora razred Lik vsebovati metodi ploscina in obseg
- prevajalnik ne ve, da kazalec lik dejansko kaže na objekt tipa Pravokotnik, Kvadrat ali Krog
- vidi samo tip kazalca (Lik)
- če v razredu Lik ne najde metod ploscina in obseg, potem klika lik.ploscina() in lik.obseg() povzročita napako

## Abstraktna metoda

- zaradi prevajalnikovih pravil morata biti metodi `ploscina` in `obseg` prisotni v razredu `Lik` ...
- ... čeprav ploščine in obsega za splošen lik ne moremo izračunati
- ... in čeprav se ne bosta nikoli klicali, saj noben lik ne bo »samo« lik (vsak bo pravokotnik, kvadrat ali krog)
- metodi zato deklariramo kot **abstraktni** (s praznim telesom)

```
public abstract double ploscina(); // ; namesto {...}  
public abstract double obseg();
```

# Abstrakten razred

- razred, čigar objektov ni mogoče ustvarjati
- če razred vsebuje abstraktno metodo, mora biti tudi sam deklariran kot abstrakten

```
public abstract class Lik {  
    public abstract double ploscina();  
    public abstract double obseg();  
}
```

- izraz `new Lik(...)` sproži napako pri prevajanju
- abstrakten razred lahko vsebuje konstruktor, vendar pa ga je mogoče klicati le iz konstruktorja podrazreda (s `super(...)`)

## Abstrakten razred

- če bi razred `Lik` lahko bil neabstrakten, bi lahko počeli take reči:

```
Lik lik = new Lik(...);  
System.out.println(lik.ploscina());
```

- klic `lik.ploscina()` bi poklical metodo `ploscina` iz razreda `Lik`
- ta pa nima telesa, zato se ne more izvesti



## Razred Lik

- v razredu Lik je smiselno definirati tudi metodo toString, saj ima izhodni niz pri vseh likih isto obliko:

*vrsta* [*podatki*]

- na primer:

*pravokotnik* [*širina* = 3.0, *višina* = 4.0]

*kvadrat* [*stranica* = 6.4]

*krog* [*polmer* = 5.7]

# Razred Lik

- podniza *vrsta* in *podatki* zagememo z metodama *vrsta* in *podatki*
- metodi *vrsta* in *podatki* morata biti prisotni v razredu *Lik*, čeprav se bodo klicale samo njune implementacije v podrazredih

```
public abstract class Lik {  
    ...  
    public String toString() {  
        return String.format("%s [%s]",  
                               this.vrsta(), this.podatki());  
    }  
  
    public abstract String vrsta();  
    public abstract String podatki();  
}
```

# Razred Lik

```
public abstract class Lik {  
    public abstract double obseg();  
    public abstract double ploscina();  
  
    public String toString() {  
        return String.format("%s [%s]",  
                               this.vrsta(), this.podatki());  
    }  
  
    public abstract String vrsta();  
    public abstract String podatki();  
}
```

- konstruktorja nam ni treba definirati, saj nimamo atributov
- prevajalnik samodejno doda privzeti konstruktor

```
public Lik() {}
```

# Razred Pravokotnik

- podrazred razreda Lik
- ni abstrakten, saj želimo ustvarjati objekte razreda Pravokotnik
- razred vsebuje
  - atributa sirina in visina
  - konstruktor
  - implementacije vseh abstraktnih metod (obvezno, ker ni abstrakten)

# Razred Pravokotnik

```
public class Pravokotnik extends Lik {  
    private double sirina;  
    private double visina;  
  
    public Pravokotnik(double sirina, double visina) {  
        this.sirina = sirina;  
        this.visina = visina;  
    }  
  
    @Override  
    public double ploscina() {  
        return this.sirina * this.visina;  
    }  
    ...  
}
```

## Razred Pravokotnik

```
public class Pravokotnik extends Lik {  
    ...  
    @Override  
    public double obseg() {  
        return 2 * (this.sirina + this.visina);  
    }  
  
    @Override  
    public String vrsta() {  
        return "pravokotnik";  
    }  
  
    @Override  
    public String podatki() {  
        return String.format("širina = %.1f, višina = %.1f",  
                               this.sirina, this.visina);  
    }  
}
```

## Razred Kvadrat

- kvadrat je poseben primer pravokotnika
- zato se nam razred Kvadrat splača izpeljati iz razreda Pravokotnik
- atributa sirina in visina se podedujeta
- metod ploscina in obseg nam ni treba redefinirati, saj se ploščina in obseg za kvadrat izračunata na enak način kot za pravokotnik
- definiramo zgolj konstruktor in redefiniramo metodi vrsta in podatki

## Konstruktor razreda Kvadrat

- objekt tipa Kvadrat ustvarimo z `new Kvadrat(stranica)`, zato konstruktor sprejme samo en parameter (*stranica*)
- oba atributa (*sirina* in *visina*) nastavimo na vrednost parametra *stranica*

```
public Kvadrat(double stranica) {  
    super(stranica, stranica);  
}
```

- Kaj se zgodi pri stavku `Kvadrat k = new Kvadrat(7.0)`?
  - ustvari se objekt tipa Kvadrat
  - pokliče se konstruktor `Kvadrat(7.0)`
  - pokliče se konstruktor `Pravokotnik(7.0, 7.0)`
  - atributa *sirina* in *visina* objekta se nastavit na 7.0



## Metodi vrsta in podatki v razredu Kvadrat

```
public class Kvadrat extends Pravokotnik {  
    ...  
    @Override  
    public String vrsta() {  
        return "kvadrat";  
    }  
  
    @Override  
    public String podatki() {  
        return String.format("stranica = %.1f", this.sirina);  
    }  
}
```

- atribut sirina v razredu Kvadrat ni neposredno dostopen
- zato izraz this.sirina v metodi podatki sproži napako pri prevajanju

## Metoda podatki v razredu Kvadrat

- **rešitev 1:** atribut sirina deklariramo kot protected
- **rešitev 2:** v razred Pravokotnik dodamo »getter« za atribut sirina

```
public class Pravokotnik {  
    ...  
    public double vrniSirino() {  
        return this.sirina;  
    }  
}
```

```
public class Kvadrat {  
    ...  
    @Override  
    public String podatki() {  
        return String.format("stranica = %.1f", this.vrniSirino());  
    }  
}
```

# Razred Krog

```
public class Krog extends Lik {  
    private double polmer;  
  
    public Krog(double polmer) {  
        this.polmer = polmer;  
    }  
  
    @Override  
    public double ploscina() {  
        return Math.PI * this.polmer * this.polmer;  
    }  
  
    @Override  
    public double obseg() {  
        return 2.0 * Math.PI * this.polmer;  
    }  
    ...  
}
```

# Razred Krog

```
public class Krog extends Lik {  
    ...  
    @Override  
    public String vrsta() {  
        return "krog";  
    }  
  
    @Override  
    public String podatki() {  
        return String.format("polmer = %.1f", this.polmer);  
    }  
}
```

## Primer zaporedja klicev metod

- Kako si sledijo klici metod pri izvedbi klica `lik.toString()`, če kazalec `lik` kaže na objekt tipa `Kvadrat`?

```
Lik lik = new Kvadrat(7.0);  
System.out.println(lik.toString());
```

1. `toString` iz razreda `Kvadrat`
2. `toString` iz razreda `Pravokotnik`
3. `toString` iz razreda `Lik`
4. `vrsta` iz razreda `Kvadrat`
5. `podatki` iz razreda `Kvadrat`
6. `vrniSirino` iz razreda `Kvadrat`
7. `vrniSirino` iz razreda `Pravokotnik`

# Glavni razred

- v glavnem razredu bomo napisali in preizkusili tri metode
- `public static void izpisiPodatke(Lik[] liki)`
  - za vsak lik v podani tabeli izpiše podatke ter ploščino in obseg
- `public static Lik likZNajvecjoPloscino(Lik[] liki)`
  - vrne kazalec na lik z največjo ploščino oziroma `null`, če je tabela prazna
- `public static Pravokotnik pravokotnikZNajvecjoSirino(Lik[] liki)`
  - vrne kazalec na pravokotnik z največjo širino oziroma `null`, če tabela ne vsebuje nobenega pravokotnika



## Metoda likZNajvecjoPloscino

- `public static Lik likZNajvecjoPloscino(Lik[] liki)`
- sprehodimo se po tabeli in za vsak lik pokličemo metodo `ploscina`
- vzdržujemo kazalec na lik z doslej največjo ploščino
- da se izognemo večkratnemu računanju ploščine, hranimo tudi doslej največjo ploščino



## Metoda likZNajvecjoPloscino

```
public static Lik likZNajvecjoPloscino(Lik[] liki) {  
    Lik najLik = null;  
    double najPloscina = 0.0;  
    for (Lik lik: liki) {  
        double ploscina = lik.ploscina();  
        if (najLik == null || ploscina > najPloscina) {  
            najPloscina = ploscina;  
            najLik = lik;  
        }  
    }  
    return najLik;  
}
```

## Metoda pravokotnikZNajvecjoSirino

- `public static Pravokotnik  
pravokotnikZNajvecjoSirino(Lik[] liki)`
- sprehodimo se po tabeli in za vsak lik preverimo, ali je pravokotnik
- če je pravokotnik, na njem pokličemo metodo `vrniSirino`
- vzdržujemo kazalec na pravokotnik z doslej največjo širino

## Metoda pravokotnikZNajvecjoSirino

```
public static Pravokotnik  
    pravokotnikZNajvecjoSirino(Lik[] liki) {  
  
    Pravokotnik naj = null;  
    for (Lik lik: liki) {  
        if (lik je pravokotnik) {  
            pridobi širino in po potrebi posodobi kazalec naj  
        }  
    }  
    return naj;  
}
```

# Preverjanje tipa objekta

- Kako ugotovimo, ali kazalec `lik` kaže na objekt tipa `Pravokotnik`?
- možnost 1: metoda `vrsta`
  - `if (lik.vrsta().equals("pravokotnik")) { ... }`
  - nize primerjamo z metodo `equals`, ne z operatorjem `==`
  - `equals` primerja vsebino
  - `==` primerja kazalca (ali gre za isti objekt)
- možnost 2: operator `instanceof`
  - `if (lik instanceof Pravokotnik) { ... }`

## Operator instanceof

- rezultat izraza `a instanceof R` je true natanko tedaj, ko `a` kaže na objekt razreda `R` ali njegovega podrazreda

```
Lik p = new Pravokotnik(3.0, 4.0);
Lik q = new Kvadrat(5.0);
Kvadrat r = new Kvadrat(6.0);

System.out.println(p instanceof Lik);    // true
System.out.println(q instanceof Lik);    // true
System.out.println(r instanceof Lik);    // true

System.out.println(p instanceof Pravokotnik); // true
System.out.println(q instanceof Pravokotnik); // true
System.out.println(r instanceof Pravokotnik); // true

System.out.println(p instanceof Kvadrat); // false
System.out.println(q instanceof Kvadrat); // true
System.out.println(r instanceof Kvadrat); // true
```

## Metoda vrsta vs. operator instanceof

- pogoj `lik.vrsta().equals("Pravokotnik")` zajame pravokotnike brez kvadratov
- pogoj `lik instanceof Pravokotnik` zajame pravokotnike in kvadrate
- če bi z `instanceof` želeli zajeti pravokotnike brez kvadratov:  
`lik instanceof Pravokotnik &&  
 !(lik instanceof Kvadrat)`

# Metoda pravokotnikZNajvecjoSirino

- prvi poskus

```
public static Pravokotnik
    pravokotnikZNajvecjoSirino(Lik[] liki) {

    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik instanceof Pravokotnik) {
            if (naj == null ||
                lik.vrniSirino() > naj.vrniSirino()) {
                naj = lik;
            }
        }
    }
    return naj;
}
```

# Metoda pravokotnikZNajvecjoSirino

- problem št. 1
  - izraz `lik.vrniSirino()` povzroči napako pri prevajanju
  - prevajalnik ne ve, da `lik` kaže na objekt tipa `Pravokotnik`
  - prevajalnik vidi samo tip kazalca (`Lik`), zato išče metodo `vrniSirino` samo v razredu `Lik`
- problem št. 2
  - izraz `naj = lik` povzroči napako pri prevajanju
  - kazalca tipa `Lik` ni mogoče prirediti spremenljivki tipa `Pravokotnik`
  - možno je samo obratno



# Pretvorba tipa

- oba problema je mogoče rešiti s pretvorbo tipa

```
public static Pravokotnik
    pravokotnikZNajvecjoSirino(Lik[] liki) {

    Pravokotnik naj = null;
    for (Lik lik: liki) {
        if (lik instanceof Pravokotnik) {
            Pravokotnik p = (Pravokotnik) lik;
            if (naj == null ||
                p.vrniSirino() > naj.vrniSirino()) {
                naj = p;
            }
        }
    }
    return naj;
}
```

# Pretvorba tipa

- recimo, da je razred B podrazred razreda A
- recimo, da je kazalec a tipa A
- pretvorba (B) a
  - prevajalnik jo vedno dovoli (nam zaupa)
  - izvajalnik sproži izjemo tipa `ClassCastException`, če kazalec a ne kaže na objekt tipa B ali njegovega podtipa

```
A a1 = new A(...);  
A a2 = new B(...);  
  
B b1 = (B) a1;    // prevede se,  
                  // izvajalnik pa sproži ClassCastException  
  
B b2 = (B) a2;    // OK
```

# Zloraba operatorja instanceof in pretvorbe tipa

- operator instanceof in pretvorba tipa omogočata programiranje v tem slogu:

```
if (lik instanceof Kvadrat) {  
    Kvadrat k = (Kvadrat) lik;  
    System.out.printf("kvadrat [stranica = %.1f]%n",  
        k.vrniSirino());  
  
} else if (lik instanceof Pravokotnik) {  
    Pravokotnik p = (Pravokotnik) lik;  
    System.out.printf("pravokotnik [širina = %.1f, višina = %.1f]%n",  
        p.vrniSirino(), p.vrniVisino());  
  
} else {  
    Krog k = (Krog) lik;  
    System.out.printf("krog [polmer = %.1f]%n", k.vrniPolmer());  
}
```

# Zloraba operatorja instanceof in pretvorbe tipa

- operator instanceof in pretvorba tipa omogočata programiranje v tem slogu:

```
if (lik instanceof Kvadrat) {  
    Kvadrat k = (Kvadrat) lik;  
    System.out.printf("kvadrat [stranica = %.1f] %n",  
        k.vrniSirino());  
  
} else if (lik instanceof Pravokotnik) {  
    Pravokotnik p = (Pravokotnik) lik;  
    System.out.printf("pravokotnik [širina = %.1f, višina = %.1f] %n",  
        p.vrniSirino(), p.vrniVisino());  
  
} else {  
    Krog k = (Krog) lik;  
    System.out.printf("krog [polmer = %.1f] %n", k.vrniPolmer());  
}
```

## Zloraba operatorja instanceof in pretvorbe tipa

- operator instanceof in pretvorba tipa naj bi se uporabljala zgolj takrat, ko je nek podatek ali izračun smiseln samo za nek podrazred
- če je podatek ali izračun smiseln za celotno hierarhijo, definiramo atribut oz. metodo v nadrazredu
- metodo redefiniramo v podrazredih, za katere se izračun razlikuje od privzetega
- če izračun v nadrazredu ni mogoč ali smiseln ali pa če izračuni za različne razrede med seboj nimajo nič skupnega, metodo v nadrazredu deklariramo kot abstraktno, v podrazredih pa jo definiramo

# Razred Object

- nadrazred vseh razredov v javi (tudi naših lastnih)
- najpomembnejše metode
  - `public String toString()`
  - `public boolean equals(Object obj)`
  - `public int hashCode()`
- vsak razred v javi samodejno vsebuje te metode in jih lahko po potrebi redefinira

## Metoda toString

- `public String toString()`
- v razredu `Object` vrne niz, ki podaja tip objekta in šestnajstiško število, ki se izračuna na podlagi pomnilniškega naslova objekta

```
Object obj = new Object();  
System.out.println(obj.toString());  
// npr. java.lang.Object@47c62251
```

- enako seveda velja za vse razrede, v katerih metoda ni redefinirana
- če metodo redefiniramo, jo ponavadi tako, da vrne niz, ki vsebuje ključne podatke objekta

## Metoda toString v (poenostavljenem) razredu Cas

```
public class Cas {  
    private int ura;  
    private int minuta;  
  
    public Cas(int h, int min) {  
        this.ura = h;  
        this.minuta = min;  
    }  
  
    @Override  
    public String toString() {  
        return String.format("%d:%02d", this.ura, this.minuta);  
    }  
}
```



## Klic metode toString

- metode `print*` imajo različico `print*(Object obj)`, ki vsebuje klic `obj.toString()`
- zato lahko klic `print*(objekt.toString())` okrajšamo v `print*(objekt)`

```
Cas cas = new Cas(10, 35);  
System.out.println(cas.toString());    // 10:35  
System.out.println(cas);               // 10:35
```

## Metoda equals

- `public boolean equals(Object obj)`
- v razredu `Object` vrne `true` natanko tedaj, ko kazalca `this` in `obj` kažeta na isti objekt


```
public class Object {  
    ...  
    public boolean equals(Object obj) {  
        return this == obj;  
    }  
}
```

- če metodo redefiniramo, jo ponavadi tako, da vrne `true` natanko tedaj, ko sta objekta enaka po vrednostih atributov
- $(a == b) \implies a.equals(b)$  (ne pa nujno obratno!)

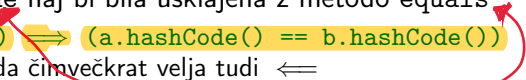
## Metoda equals v razredu Cas

- podobna metodi jeEnakKot, a ne povsem enaka
- `public boolean jeEnakKot(Cas drugi)`
- `public boolean equals(Object drugi)`
- če kazalca `this` in `drugi` kažeta na isti objekt, vrnemo `true`
- sicer preverimo, ali kazalec `drugi` kaže na objekt tipa `Cas`
  - če ne, vrnemo `false`
- tip kazalca `drugi` lahko sedaj varno pretvorimo v `Cas`
- primerjamo objekta po atributih

## Metoda equals v razredu Cas

```
@Override
public boolean equals( Object drugi) {
    if (this == drugi) {
        return true;
    }
    if (!(drugi instanceof Cas)) {
        return false;
    }
    Cas drugiCas = (Cas) drugi;
    return (this.ura == drugiCas.ura &&
        this.minuta == drugiCas.minuta);
}
```

# Metoda hashCode

- `public int hashCode()`
  - v razredu `Object` vrne število, ki se izračuna na podlagi pomnilniškega naslova objekta
  - če metodo redefiniramo, jo ponavadi tako, da vrne število, ki čimbolj enolično določa objekt
  - metoda `hashCode` naj bi bila usklajena z metodo `equals`
    - `a.equals(b)`  $\Rightarrow$  `(a.hashCode() == b.hashCode())`
    - zaželeno je, da čimvečkrat velja tudi  $\Leftarrow$
- 

# Metoda hashCode

- rezultat metode hashCode ponavadi izračunamo kot

$$\underline{h = p_1 h(a_1) + p_2 h(a_2) + \dots + p_n h(a_n)}$$

- $p_1, \dots, p_n$  so praštevila ?
- $h(a_1), \dots, h(a_n)$  so rezultati metode hashCode za posamezne attribute
- za attribute primitivnih tipov

- int:  $h(a_i) = \text{Integer.hashCode}(a_i)$
- double:  $h(a_i) = \text{Double.hashCode}(a_i)$
- char:  $h(a_i) = \text{Character.hashCode}(a_i)$
- ...

t


## Metoda hashCode v razredu Cas

```
@Override
public int hashCode() {
    return (17 * Integer.hashCode(this.ura) +
           31 * Integer.hashCode(this.minuta));
}
```

## Razred Object in tabele

- če je tabela tipa `Object[]`, lahko vsebuje kazalce na objekte poljubnih tipov

```
Object[] mesanica = {  
    new Oseba("Jože", "Gorišek", 'M', 1953),  
    new Cas(10, 35),  
    "Dober dan!"  
};  
  
for (Object element: mesanica) {  
    System.out.println(element);  
}
```



```
Jože Gorišek (M), 1953  
10:35  
Dober dan!
```



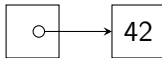
## Primitivni in ovojni tipi

- primitivni tipi niso del hierarhije javanskih razredov
- tabela tipa `Object[]` ne more vsebovati elementov tipa `int`, `double` itd.
- namesto tega lahko uporabimo **ovojne tipe** Integer, Double itd.

```
int a = 42;  
Integer b = Integer.valueOf(42);
```



a



b

## Primitivni in ovojni tipi

primitivni tip	ovojni tip
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

# Pretvarjanje med primitivnimi in ovojnimi tipi

- `int a = ...;`  
`Integer b = ...;`
- osnovni način
  - `int`  $\rightarrow$  `Integer`: `Integer c = Integer.valueOf(a);`
  - `Integer`  $\rightarrow$  `int`: `int d = b.intValue();`
- samodejno pretvarjanje (angl. auto(un)boxing)
  - `int`  $\rightarrow$  `Integer`: `Integer c = a;`
  - `Integer`  $\rightarrow$  `int`: `int d = b;`

# Pretvarjanje med primitivnimi in ovojnimi tipi

- samodejno pretvarjanje nam omogoča take reči ...

```
Object[] mesanica = {  
    "Dober dan!",  
    42,          // Integer.valueOf(42)  
    -3.14,       // Double.valueOf(-3.14)  
    true,        // Boolean.valueOf(true)  
    'a'          // Character.valueOf('a')  
};  
  
for (Object element: mesanica) {  
    System.out.println(element);  
}
```

```
Dober dan!  
42  
-3.14  
true  
a
```

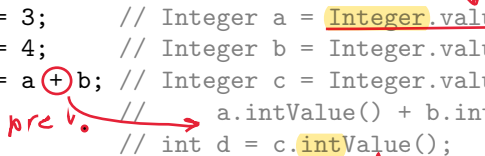


print

## Pretvarjanje med primitivnimi in ovojnimi tipi

- prevajalnik sam doda klice pretvornih metod

```
Integer a = 3;      // Integer a = Integer.valueOf(3);  
Integer b = 4;      // Integer b = Integer.valueOf(4);  
Integer c = a + b;  // Integer c = Integer.valueOf(  
                    // a.intValue() + b.intValue());  
int d = c;          // int d = c.intValue();
```



## Primitivni in ovojni tipi

- primitivni in ovojni tipi kljub vsemu niso povsem enakovredni
- spremenljivka ovojnega tipa lahko ima tudi vrednost null

```
int a = 3;
System.out.println(a + 1);    // 4

Integer b = 5;
System.out.println(b + 1);    // 6

int[] t = new int[3];        // {0, 0, 0}
System.out.println(t[0] + 1); // 1

Integer[] u = new Integer[3]; // {null, null, null}
System.out.println(u[0] + 1);  // NullPointerException
                               // (zaradi u[0].intValue())
```

*Handwritten annotations:*

- Red arrow from `b + 1` to `b.intValue() = 76`
- Red text `[0, 0, 0]` with an arrow pointing to `t[0]`
- Red arrow from `u[0]` to `NullPointerException`

# Vektor z elementi poljubnega referenčnega tipa

## Tabela tipa object

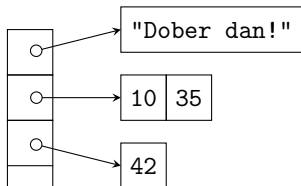
- tabelo elementov deklariramo kot Object[] elementi
- metode



```
public Object vrni(int indeks) { ... }  
public void nastavi(int indeks, Object vrednost) { ... }  
public void dodaj(Object vrednost) { ... }  
public void vstavi(int indeks, Object vrednost) { ... }  
public void odstrani(int indeks) { ... }  
public String toString() { ... }
```

## Vektor z elementi poljubnega referenčnega tipa

```
Vektor vektor = new Vektor();  
vektor.dodaj("Dober dan!");  
vektor.dodaj(new Cas(10, 35));  
vektor.dodaj(42);  
Object obj0 = vektor.vrni(0);  
Object obj1 = vektor.vrni(1);  
Object obj2 = vektor.vrni(2);
```



- kazalec obj0 je tipa Object, kaže pa na objekt tipa String
- zato lahko obj0 pretvorimo v tip String
- podobno velja za obj1 in obj2

```
String s = (String) obj0;  
Cas c = (Cas) obj1;  
Integer n = (Integer) obj2;
```




# Vektor z elementi poljubnega referenčnega tipa

- uporabnik vektorja mora poznati tipe objektov, shranjenih v vektorju
- napačna pretvorba tipa sproži `ClassCastException`

```
Vektor vektor = new Vektor();  
vektor.dodaj("Dober dan!")  
Integer n = (Integer) vektor.vrni(0); // ClassCastException
```

- boljša rešitev: `generiki`

# Slovar

- 
- podatkovna struktura, ki preslikuje ključe v vrednosti
  - posplošitev tabele
    - tabela: ključi so cela števila (indeksi)
    - slovar: ključi so lahko poljubni objekti

# Konstruktor in metodi

- `public Slovar()`
  - ustvari prazen slovar
- `public void shrani(Object kljuc, Object vrednost)`
  - če slovar ne vsebuje podanega ključa, potem metoda shrani vanj nov par ključ-vrednost, sicer pa zamenja vrednost, povezano s podanim ključem
- `public Object vrni(Object kljuc)`
  - vrne vrednost, ki pripada podanemu ključu, oziroma `null`, če slovar ne vsebuje podanega ključa

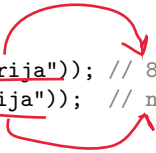
# Slovar

- **primer:** ključ je država, vrednost pa število njenih sosed

```
Slovar drzava2sosedje = new Slovar();
drzava2sosedje.shrani("Slovenija", 4);
drzava2sosedje.shrani("Avstrija", 8);
drzava2sosedje.shrani("Češka", 4);
drzava2sosedje.shrani("Francija", 8);
drzava2sosedje.shrani("Italija", 6);
drzava2sosedje.shrani("Slovaška", 5);
drzava2sosedje.shrani("Švica", 5);

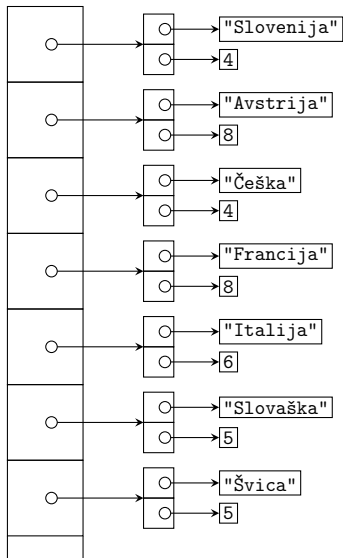
System.out.println(drzava2sosedje.vrni("Avstrija")); // 8
System.out.println(drzava2sosedje.vrni("Nemčija")); // null

drzava2sosedje.shrani("Avstrija", 9);
System.out.println(drzava2sosedje.vrni("Avstrija")); // 9
```



# Naivna implementacija slovarja

- tabela parov ključ-vrednost
- tabela se po potrebi »raztegne« (kot pri vektorju)
- nove pare ključ-vrednost dodajamo na konec tabele
- ključe iščemo s sprehtodom po tabeli



# Implementacija slovarja z zgoščeno tabelo

- sprehod po tabeli je neučinkovit
- boljši pristop: indeks celice, v katero shranimo par ključ-vrednost, izračunamo na podlagi vrednosti `kljuc.hashCode()`
- vrednost `kljuc.hashCode()` preslikamo na interval `[0, tabela.length - 1]`
- uporabimo ostanek pri deljenju (in upoštevamo možnost, da je vrednost `kljuc.hashCode()` negativna)

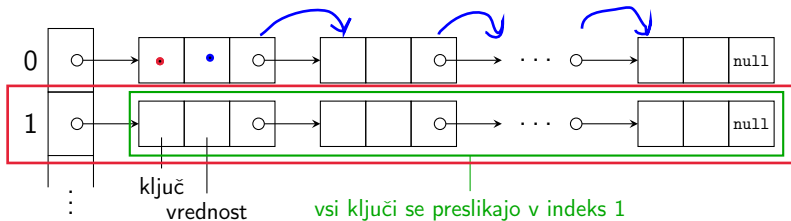
zaradi negativnih indeksov

```
private int indeks(Object kljuc) {  
    int n = this.tabela.length;  
    return ((kljuc.hashCode() % n) + n) % n;  
}
```

- **problem:** lahko se več ključev preslika v isto celico

# Implementacija slovarja z zgoščeno tabelo

- **rešitev**: celica z indeksom  $i$  kaže na seznam vseh ključev (in pripadajočih vrednosti), za katere metoda `indeks` vrne vrednost  $i$
- seznam implementiramo s **povezanim seznamom**



- celotno strukturo (tabelo in povezane sezname) imenujemo **zgoščena tabela**

## Implementacija slovarja z zgoščeno tabelo

- povezani sezname so sestavljeni iz objektov tipa Vozlisce

```
// notranji razred v razredu Slovar
private static class Vozlisce {
    Object kljuc;
    Object vrednost;
    Vozlisce naslednje;
}
```

staticni notranji razredi

- element tabele z indeksom  $i$  kaže na prvi objekt tipa Vozlisce, pri katerem se ključ preslika v indeks  $i$
- atribut naslednje kaže na naslednji objekt tipa Vozlisce, pri katerem se ključ preslika v isti indeks
- če ni naslednika, potem atribut naslednje vsebuje null

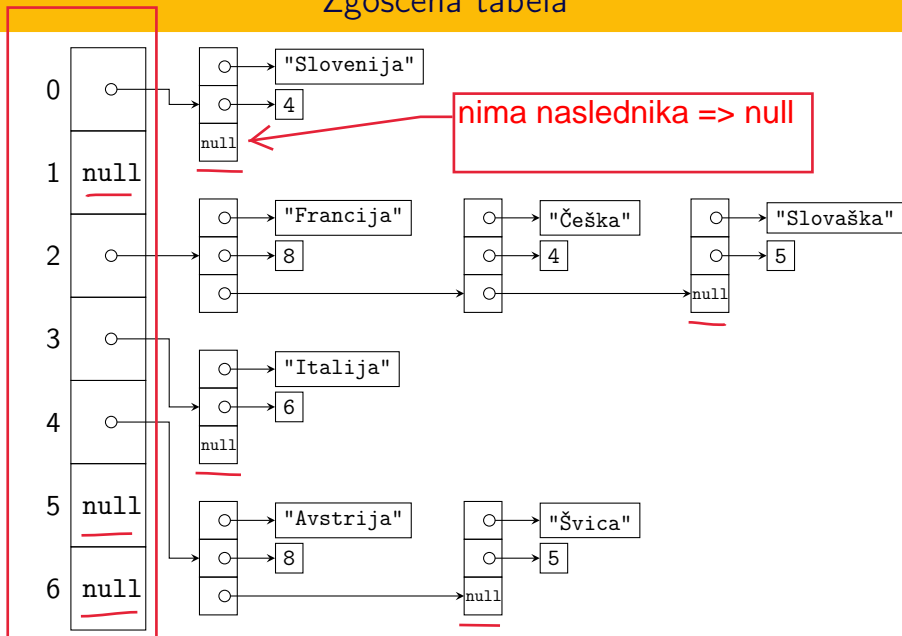


## Države in sosede

- recimo, da je dolžina tabele enaka 7
- indeksi, v katere se preslikajo posamezni ključi
  - "Slovenija".hashCode()  $\rightarrow -1541319689 \xrightarrow{\%7} 0$
  - "Avstrija".hashCode()  $\rightarrow -1927211996 \xrightarrow{\%7} -3 \rightarrow 4$

ključ	indeks
Slovenija	0
Avstrija	4
Češka	2
Italija	3
Francija	2
Slovaška	2
Švica	4

## Zgoščena tabela



## Atributi razreda Slovar

- tabela: tabela kazalcev na začetke povezanih seznamov
- VELIKOST\_TABELE: privzeta velikost te tabele

```
public class Slovar {  
    ...  
    private static final int VELIKOST_TABELE = 97;  
    private Vozlisce[] tabela;  
    ...  
}
```

## Konstruktorja razreda Slovar

- konstruktorja zgolj izdelata tabelo privzete ali podane velikosti

```
public Slovar() {  
    this(VELIKOST_TABELE);  
}  
  
public Slovar(int velikostTabele) {  
    this.tabela = new Vozlisce[velikostTabele];  
}
```

# Iskanje ključa

- izračunamo indeks ključa v tabeli
- sprehodimo se po povezanem seznamu, ki vsebuje ključe z istim indeksom
- iskani ključ primerjamo s ključi, shranjenimi v posameznih vozliščih
  - uporabimo metodo equals

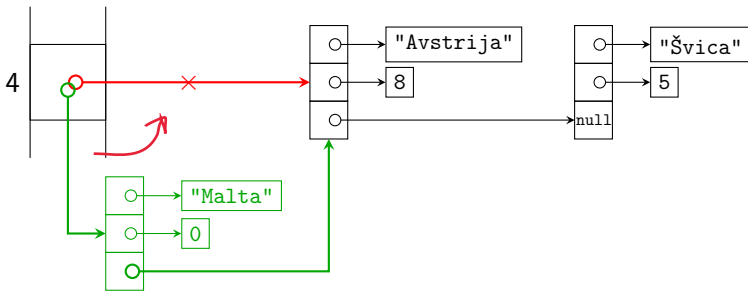
```
private Vozlisce poisci(Object kljuc) {  
    int indeks = this.indeks(kljuc);  
    Vozlisce vozlisce = this.tabela[indeks];  
    while (vozlisce != null && !vozlisce.kljuc.equals(kljuc)) {  
        vozlisce = vozlisce.naslednje;  
    }  
    return vozlisce;  
}
```

## Vračanje vrednosti, povezane s podanim ključem

```
public Object vrni(Object kljuc) {  
    Vozlisce vozlisce = this.poisci(kljuc);  
    if (vozlisce == null) {  
        return null;  
    }  
    return vozlisce.vrednost;  
}
```

## Shranjevanje parov ključ-vrednost

- preverimo, ali vozlišče s podanim ključem že obstaja
- če obstaja, samo posodobimo vrednost
- sicer ustvarimo novo vozlišče in ga vstavimo na začetek povezanega seznama



## Shranjevanje parov ključ-vrednost

```
public void shrani(Object kljuc, Object vrednost) {  
    Vozlisce vozlisce = this.poisici(kljuc);  
    if (vozlisce != null) {  
        vozlisce.vrednost = vrednost;  
    } else {  
        int indeks = this.indeks(kljuc);  
        vozlisce = new Vozlisce(kljuc, vrednost,  
                                this.tabela[indeks]);  
        this.tabela[indeks] = vozlisce;  
    }  
}
```



## Primer uporabe slovarja: telefonski imenik

- program, ki v zanki bere imena in bere oz. izpisuje pripadajoče telefonske številke
- če ime še ni znano, prebere telefonsko številko
- v nasprotnem primeru izpiše tel. številko, ki pripada imenu

Vnesite ime: Janez

Oseba Janez še ni shranjena v imeniku.

Vnesite telefonsko številko: 041 234 567

Vnesite ime: Mojca

Oseba Mojca še ni shranjena v imeniku.

Vnesite telefonsko številko: 09 876 543

Vnesite ime: Janez

Telefonska številka: 041 234 567

Vnesite ime: Mojca

Telefonska številka: 09 876 543

# Telefonski imenik

- ključ: ime
- vrednost: telefonska številka
- za vsako prebrano ime preverimo, ali ključ s to vsebino že obstaja v slovarju
- če obstaja, vrnemo pripadajočo vrednost
- sicer preberemo telefonsko številko in jo shranimo skupaj z imenom (ime postane ključ, številka pa vrednost)

# Telefonski imenik

```
Slovar ime2stevilka = new Slovar();
Scanner sc = new Scanner(System.in);
System.out.print("Vnesite ime: ");
String ime = sc.nextLine();
while (ime.length() > 0) {
    String stevilka = (String) ime2stevilka.vrni(ime);
    if (stevilka == null) {
        System.out.printf("Oseba %s še ni shranjena v imeniku.%n",
                           ime);
        System.out.print("Vnesite telefonsko številko: ");
        stevilka = sc.nextLine();
        ime2stevilka.shrani(ime, stevilka);
    } else {
        System.out.printf("Telefonska številka: %s%n", stevilka);
    }
    System.out.println();
    System.out.print("Vnesite ime: ");
    ime = sc.nextLine();
}
```