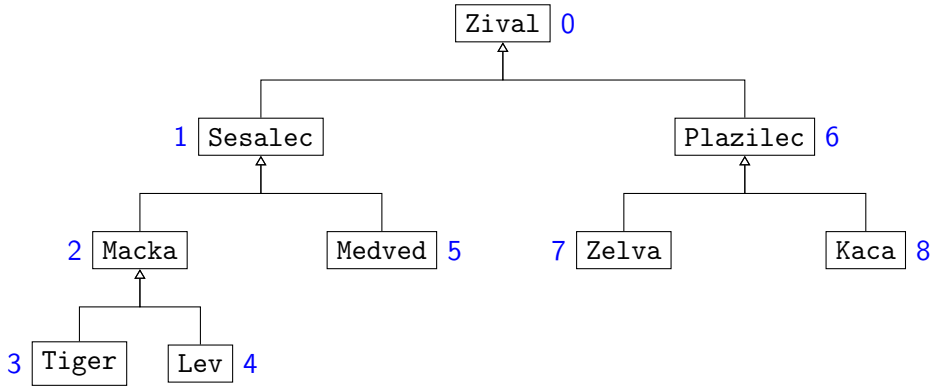
Rešitev devete domače naloge (Divjina)

Pri tej nalogi nam bo koristilo, če bomo tipom priredili indekse — števila od 0 do 8. To bomo storili tako, kot prikazuje slika 1.



Slika 1: *Indeksi razredov v hierarhiji.*

V razredu `Zival` si pripravimo enodimenzionalno tabelo, v kateri bomo vzdrževali število primerov, ko se žival določenega tipa hrani, in dvodimenzionalno tabelo, v kateri bomo vzdrževali število primerov, ko žival določenega tipa preganja žival nekega drugega (ali istega) tipa:

```
public class Zival {
    private static final int[] ST_HRANJENJ = new int[9];
    private static final int[][] ST_PREGANJANJ = new int[9][9];
    ...
}
```

V celici `ST_HRANJENJ[`i`]` bomo vzdrževali število primerov, ko se prehranjuje žival tipa z indeksom i ali podtipa tega tipa. Na primer, v celici `ST_HRANJENJ[1]` bomo vzdrževali število primerov, ko se hrani sesalec (vključno s primeri, ko se hrani mačka, tiger, lev ali medved). V celici `ST_PREGANJANJ[`i`][`j`]` bomo vzdrževali število primerov, ko žival tipa z indeksom i ali podtipa tega tipa preganja žival tipa z indeksom j ali podtipa tega tipa. Tabeli sta statični, ker nista vezani na konkreten objekt tipa `Zival`, ampak pripadata celotnemu razredu. Hm ... zakaj smo dodali ločilo `final`, če pa bomo tabeli spreminjali? Pozor: spreminjali bomo elemente tabel, ne pa spremenljivk `ST_HRANJENJ` in `ST_PREGANJANJ` (ki sta, kot vemo, kazalca na tabeli).

Od tod naprej vodi veliko poti. Ubrali bomo štiri.

Prva rešitev

Pri tej rešitvi si pomagamo z metodo `indeksiTipov`, ki vrne tabelo, ki vsebuje indeks tipa objekta `this` in indekse vseh nadtipov tega tipa.

```
public class Zival {
    ...
    private int[] indeksiTipov() {
        if (this instanceof Tiger) {
            return new int[]{3, 2, 1, 0};
```

```
        }
        if (this instanceof Lev) {
            return new int[]{4, 2, 1, 0};
        }
        if (this instanceof Macka) {
            return new int[]{2, 1, 0};
        }
        if (this instanceof Medved) {
            return new int[]{5, 1, 0};
        }
        if (this instanceof Sesalec) {
            return new int[]{1, 0};
        }
        if (this instanceof Zelva) {
            return new int[]{7, 6, 0};
        }
        if (this instanceof Kaca) {
            return new int[]{8, 6, 0};
        }
        if (this instanceof Plazilec) {
            return new int[]{6, 0};
        }
        if (this instanceof Zival) {
            return new int[]{0};
        }
        throw new RuntimeException("Nekaj smrdi!");
    }
    ...
}
```

Na primer, če objekt, na katerega kaže kazalec `this`, pripada tipu `Medved`, bo metoda vrnila tabelo z indeksi tipov `Medved`, `Sesalec` in `Zival`. Vrstni red preverjanj je pomemben: ker, denimo, iz resničnosti izraza `this instanceof Tiger` sledi resničnost izraza `this instanceof Macka`, najprej preverimo, ali je žival `this` tiger, in šele zatem preverimo, ali je mačka. Če bi pogojna stavka med seboj zamenjali, tigra sploh ne bi zaznali kot objekta tipa `Tiger`, ampak zgolj kot objekt tipa `Macka`.

Metoda `seHrani` pokliče metodo `indeksiTipov` nad objektom `this` in elemente tabele `ST_HRANJENJ` na indeksih, ki jih vrne ta metoda, poveča za 1. Če je objekt `this` tipa `Medved`, bo metoda torej povečala elemente na indeksih 5 (`Medved`), 1 (`Sesalec`) in 0 (`Zival`).

```
public class Zival {
    ...
    public void seHrani() {
        int[] indeksi = this.indeksiTipov();
        for (int i: indeksi) {
            ST_HRANJENJ[i]++;
        }
    }
    ...
}
```

Metoda `preganja` poveča elemente `ST_PREGANJANJ[`i`][`j`]` za vse take pare (i, j), da je i indeks tipa objekta `this` ali nekega njegovega (neposrednega ali posrednega) nadtipa, j pa indeks tipa objekta `druga` ali nekega njegovega (neposrednega ali posrednega) nadtipa.

```
public class Zival {
    ...
    public void preganja(Zival druga) {
        int[] thisIndeksi = this.indeksiTipov();
        int[] drugaIndeksi = druga.indeksiTipov();
        for (int i: thisIndeksi) {
            for (int j: drugaIndeksi) {
                ST_PREGANJANJ[i][j]++;
            }
        }
    }
    ...
}
```

Metoda `steviloHranjenj` zgolj vrne element `ST_HRANJENJ[`i`]`, kjer je i indeks tipa objekta `this`. Metodo `indeksiTipov` smo napisali tako, da je ta indeks v vrnjeni tabeli vedno na prvem mestu. Metoda `steviloPreganjanj` deluje na enak način.

```
public class Zival {
    ...
    public int steviloHranjenj() {
        return ST_HRANJENJ[this.indeksiTipov()[0]];
    }

    public int steviloPreganjanj(Zival druga) {
        return ST_PREGANJANJ[this.indeksiTipov()[0]][druga.indeksiTipov()[0]];
    }
}
```

Vsi ostali razredi so prazni, saj nam v njih ni treba redefinirati nobene metode, privzeti konstruktorji pa prav tako povsem zadoščajo.

```
public class Sesalec extends Zival { }

public class Macka extends Sesalec { }

public class Tiger extends Macka { }

public class Lev extends Macka { }

public class Medved extends Sesalec { }

public class Plazilec extends Zival { }

public class Zelva extends Plazilec { }

public class Kaca extends Plazilec { }
```

Druga rešitev

Operator `instanceof` nam ni pretirano všeč, saj je rešitev zaradi njega manj elegantna in prilagodljiva, kot bi lahko bila. Odpovemo se mu lahko tako, da izkoristimo že zgrajeno hierarhijo razredov. Takole bi šlo:

- V hierarhiji definiramo metodo

  `public int indeks()`,

  ki vrne indeks tipa objekta `this`. Na primer, v razredu `Medved` je metoda redefinirana tako, da vrne število 5.

- V hierarhiji definiramo metodo

  `public Zival nadobjekt()`,

  ki vrne objekt neposrednega nadtipa tipa, ki mu pripada objekt `this`. Na primer, v razredu `Medved` je metoda redefinirana tako, da vrne objekt tipa `Sesalec`. Izjema je razred `Zival`, saj metoda v njem ne vrne objekta tipa `Object`, ampak preprosto `null`.

```
public class Zival {
    ...
    public int indeks() {
        return 0;
    }

    public Zival nadobjekt() {
        return null;
    }
}

public class Sesalec extends Zival {
    @Override
    public int indeks() {
        return 1;
    }

    @Override
    public Zival nadobjekt() {
        return new Zival();
    }
}

public class Tiger extends Macka {
    @Override
    public int indeks() {
        return 3;
    }

    @Override
    public Zival nadobjekt() {
        return new Macka();
    }
```

```
}

public class Lev extends Macka {
    @Override
    public int indeks() {
        return 4;
    }

    @Override
    public Zival nadobjekt() {
        return new Macka();
    }
}

public class Medved extends Sesalec {
    @Override
    public int indeks() {
        return 5;
    }

    @Override
    public Zival nadobjekt() {
        return new Sesalec();
    }
}

public class Plazilec extends Zival {
    @Override
    public int indeks() {
        return 6;
    }

    @Override
    public Zival nadobjekt() {
        return new Zival();
    }
}

public class Zelva extends Plazilec {
    @Override
    public int indeks() {
        return 7;
    }

    @Override
    public Zival nadobjekt() {
        return new Plazilec();
    }
}

public class Kaca extends Plazilec {
```

```
    @Override
    public int indeks() {
        return 8;
    }

    @Override
    public Zival nadobjekt() {
        return new Plazilec();
    }
}
```

Metodo `seHrani` lahko sedaj napišemo tako, da s pomožnim kazalcem `p` najprej pokažemo na objekt `this`, potem pa v zanki izvajamo stavek

`p = p.nadobjekt();`

dokler ne dobimo `null`. Vsakokrat pridobimo indeks tipa trenutnega objekta in povečamo element tabele `ST_HRANJENJ` na tem indeksu.

```
public class Zival {
    ...
    public void seHrani() {
        Zival p = this;
        while (p != null) {
            ST_HRANJENJ[p.indeks()]++;
            p = p.nadobjekt();
        }
    }
    ...
}
```

Na primer, če metodo `seHrani` pokličemo nad objektom tipa `Medved`, bo kazalec `p` najprej kazal na objekt tipa `Medved` (indeks bo enak 5), nato na objekt tipa `Sesalec` (indeks bo enak 1), nazadnje pa na objekt tipa `Zival` (indeks bo enak 0). Metoda `preganja` deluje na enak način, le da potrebujemo vgnezdeno zanko in dva kazalca.

```
public class Zival {
    ...
    public void preganja(Zival druga) {
        Zival p = this;
        while (p != null) {
            Zival q = druga;
            while (q != null) {
                ST_PREGANJANJ[p.indeks()][q.indeks()]++;
                q = q.nadobjekt();
            }
            p = p.nadobjekt();
        }
    }
    ...
}
```

Metodi `steviloHranjenj` in `steviloPreganjanj` sta kratki in jedrnati:

```
public class Zival {
    ...
    public int steviloHranjenj() {
        return ST_HRANJENJ[this.indeks()];
    }

    public int steviloPreganjanj(Zival druga) {
        return ST_PREGANJANJ[this.indeks()][druga.indeks()];
    }
    ...
}
```

Tretja rešitev

Druga rešitev je elegantnejša od prve, saj nam omogoča, da v hierarhijo enostavno dodamo nov razred. Recimo, če bi dodali razred `Pes` kot podrazred razreda `Sesalec`, bi morali spremeniti le velikost tabel `ST_HRANJENJ` in `ST_PREGANJANJ`, pa še temu bi se lahko ognili, če bi namesto tabel uporabili kaj bolj prilagodljivega.

Vendar pa tudi druga rešitev ni popolna, saj, denimo, za vsak klic metode `seHrani` nad objektom tipa `Tiger` po nepotrebnem ustvari tri nove objekte (tipa `Macka`, `Sesalec` in `Zival`), še več objektov pa se lahko ustvari pri klicu metode `preganja`. Tej nadlogi se k sreči dokaj zlahka izognemo: namesto da ob vsakem klicu metode `nadobjekt` vrnemo nov objekt, vrnemo *obstoječi* objekt, ki ga ustvarimo kot statični atribut v pripadajočem razredu.

```
public class Zival {
    ...
    protected static final Zival OBJEKT = new Zival();
    ...
}

public class Sesalec extends Zival {
    ...
    protected static final Zival OBJEKT = new Sesalec();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}

public class Macka extends Sesalec {
    ...
    protected static final Zival OBJEKT = new Macka();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
```

```
}

public class Tiger extends Macka {
    ...
    protected static final Zival OBJEKT = new Tiger();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}

public class Lev extends Macka {
    ...
    protected static final Zival OBJEKT = new Lev();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}

public class Medved extends Sesalec {
    ...
    protected static final Zival OBJEKT = new Medved();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}

public class Plazilec extends Zival {
    ...
    protected static final Zival OBJEKT = new Plazilec();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}

public class Zelva extends Plazilec {
    ...
    protected static final Zival OBJEKT = new Zelva();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}
```

```
public class Kaca extends Plazilec {
    ...
    protected static final Zival OBJEKT = new Kaca();
    ...
    @Override
    public Zival nadobjekt() {
        return super.OBJEKT;
    }
}
```

Četrta rešitev

Morda ste se že vprašali, zakaj moramo v vsakem razredu redefinirati metodo `nadobjekt`, če pa je njena vsebina povsod enaka (razen v razredu `Zival`):

```
public Zival nadobjekt() {
    return super.OBJEKT;
}
```

Zakaj ne bi metode definirali samo v razredih `Sesalec` in `Plazilec`, v ostalih pa bi jo preprosto podedovali? Žal to ni mogoče. Beseda `super` se namreč pri podedovanih metodah nanaša na isti razred kot v matičnih metodah. Na primer, pri podanih razredih

```
public class A {
    public void f() {
        System.out.println("A");
    }
}

public class B extends A {
    @Override
    public void f() {
        super.f();
        System.out.println("B");
    }
}

public class C extends B {
    @Override
    public void f() {
        super.f();
        System.out.println("B");
    }
}
```

bi stavek

```
new C().f();
```

po pričakovanjih izpisal

```
A
B
B
```

če pa metode `f` v razredu `C` ne bi redefinirali, bi dobili izpis

```
A
B
```

saj bi se beseda `super` v metodi `f`, ki bi jo razred `C` podedoval od razreda `B`, še vedno nanašala na razred `A`, ne pa na razred `B`.

Rešitev dejansko obstaja, a ni najbolj elegantna. Redefiniranih metod se lahko znebimo z uporabo *refleksije*, javinega podsistema, ki nam omogoča dostop do elementov objektov in razredov mimo običajnih pravil.

Refleksija je močna, a tudi nevarna, saj nam omogoča, da zaobidemo zaščito, ki jo nudi določilo `private`. Uporabo refleksije zato odsvetujemo (razen v izjemnih primerih); če ne drugega, je koda, ki uporablja refleksijo, bistveno manj pregledna od kode, ki do elementov objektov in razredov dostopa na uveljavljeni način.

Vrnimo se k naši nalogi. Če v razredu `Zival` metodo `nadobjekt` definiramo takole ...

```
public Zival nadobjekt() {
    Class nadrazred = this.getClass().getSuperclass();
    try {
        return (Zival) nadrazred.getDeclaredField("OBJEKT").get(null);
    } catch (NoSuchFieldException | IllegalAccessException ex) {}
    return null;
}
```

... potem nam je v podrazredih ni treba redefinirati. Razlago pa poiščite sami ...