Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

PAZ1C

Róbert Novotný
robert.novotny@upjs.sk

30. 9. 2009

V predchádzajúcej časti ste videli...

- voľba vhodnej reprezentácie dát je dôležitá
  - zvážme životnosť nášho programu
  - efektivitu programovania
  - výkonnosť
- String-y používame len na to, na čo boli určené
- trieda je často najvhodnejšou podobou

Triedy = stav + schopnosti

stav	schopnosti
to, čo si inštancia pamätá	to, čo inštancia dokáže
popisované podstatnými / prídavnými menami vek, rasa, počet valcov, absolvované predmety, naštartovaný	popisované slovesami / rozkazmi nastav vek! zapni sa! naštartuj sa! zapíš predmet!
dáta	algoritmy
inštančné premenné	metódy

Ukážka metódy

Pes.java

```
class Pes {
  String rasa;
  int vek;
  void stekaj() {
      System.out.println("Haf!");
  }
}
```

PesTester.java

```
// domyslíme si public class a main(...
Pes dunčo = new Pes();
dunčo.štekaj();
```

Využívame (psychické) stavy psa

- v metódach môžeme veselo využívať stavové premenné.

Pes.java

```
class Pes {
   String rasa;
   int vek;
   void stekaj() {
       if (vek < 1) {
              System.out.println("Píp!");
       } else {
              System.out.println("Haf!");
       }
   }
}
```

Schopnosť môže meniť stav


```
class Pes {
   String rasa;
   int vek;

   void pridajRok() {
       vek = vek + 1;
   }
}
```

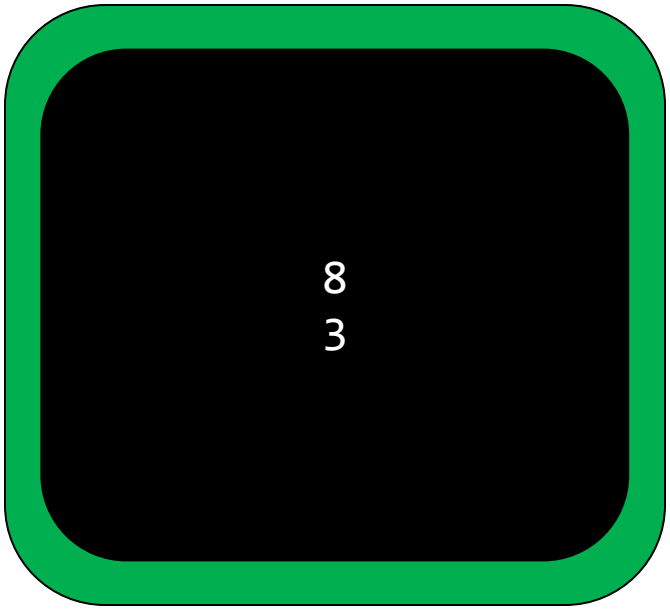
```
public static void main(String[] args) {
   Pes rex = new Pes();
   rex.vek = 7;
   System.out.println(rex.vek);
   pes.pridajVek();
   System.out.println(rex.vek);
}
```

```
7
8
```

Každá inštancia má svoj vlastný stav!

- Každý objekt má svoj vlastný stav
  - Dunčo má 7 rokov, Lajka 3
  - Zmena stavu ovplyvní len konkrétny objekt, ostatné nie!

```
public static void main(String[] args) {
   Pes dunčo = new Pes();
   dunčo.vek = 7;
   Pes lajka = new Pes();
   lajka.vek = 3
   dunčo.pridajVek();
   System.out.println(dunčo.vek);
   System.out.println(lajka.vek);
}
```



Metóda môže vracať aj hodnotu

návratový typ

```
String stekaj() {
   if (vek < 1) {
       return "Píp!";
   } else {
       return "Haf!";
   }
}
```


```
Pes dunčo = new Pes();
dunčo.vek = 27;
String štek = dunčo.stekaj();
System.out.println(štek);
```

Zložitejšie štruktúry

- minule sme videli, že
  s jednoduchými typmi sa ďaleko nedostaneme
- potrebujeme triedy
- čo ak potrebujeme niekam ukladať viacero inštancií?
  - zoznam študentov
  - množina pív
  - ...

Zložitejšie štruktúry

- Komplexné štruktúry potrebujeme reprezentovať aj v matematike
  - **množiny**

    „Teória množín je programovací jazyk.“
  - **zoznamy**
    - **zoznamy zoznamov** (matice)
  - **funkcie** (postupnosti)
  - **stromy** (grafy)

Polia obyčajné

„...a od Prešova v tym poľu"

- deklarácia a inicializácia

```
String[] reťazce = new String[4];
```

- alternatívna deklarácia a inicializácia

```
String[] reťazce = { "Ferko", "Miško" };
```

- prístup k prvkom poľa

```
String prvý = reťazce[0];
```

- dĺžka poľa

```
String dlzka = reťazce.length;
```

pole dĺžky 4

pole dĺžky 2

v kučeravých zátvorkách vymenujeme prvky poľa

bez zátvoriek! toto nie je metóda

Výhody a nevýhody polí

- dĺžka poľa môže byť určená premennou
  - existujú jazyky (Pascal), kde musí byť dĺžka určená konštantou
- netreba zabúdať, že na začiatku sú v políčkach nuly
- lenže veľkosť poľa sa už nemôže meniť
- komplikované vkladanie prvkov do existujúceho poľa
  - nadeklarovať nové pole o jedna dlhšie
  - v cykle skopírovať prvú časť poľa, nový prvok, zvyšok
- komplikované mazanie prvkov z poľa
  - nadeklarovať nové pole o jedna kratšie
  - v cykle skopírovať prvú časť poľa, zvyšok za vymazaným prvkom

Polia dynamické čili zoznamy

- našťastie máme zoznamy – trieda **java.util.ArrayList**
- deklarácia a inicializácia

```
java.util.ArrayList<String> mená
    = new java.util.ArrayList<String>();
```

- pridanie prvku (na koniec zoznamu)

```
mená.add("Rex");
```

- prístup k prvku

```
String meno = mená.get(0);
```

- dĺžka zoznamu

```
int dĺžka = mená.size();
```

po importe môžeme vynechať `java.util.`

so zátvorkami!
toto je metóda!

Polia dynamické čili zoznamy

- odstránenie prvku

  `mená.remove(2);`

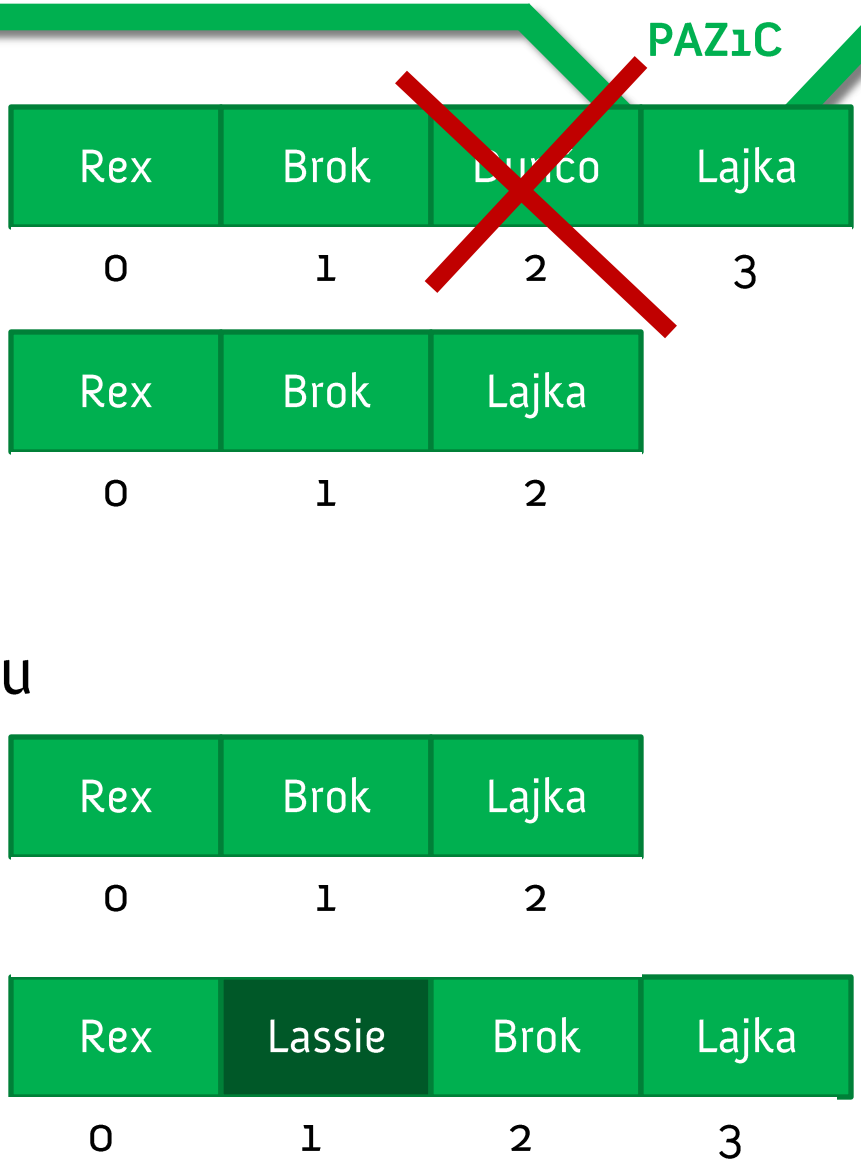
  – prvky sa posunú doľava
- pridanie prvku na pozíciu

  `mená.add(1, "Lassie");`

  – pridanie na „druhú" pozíciu

  – prvky sa posunú doprava
- ostatné operácie

  – viď dokumentácia



Polia vs. zoznamy

PAZ1C

- polia vieme previesť na zoznam a späť
- zoznam => pole

```
String[] poleMien
  = mená.toArray(new String[0]);
```

potrebujeme sem dať prázdne pole

- pole => zoznam

```
java.util.List<String> mená
  = java.util.Arrays.asList(poleMien);
```

- porovnanie
  - prístup k poliam je rýchlejší – chlieviky v pamäti za sebo

Vo všeobecnosti:
Zoznamy majú všetky výhody polí a odstraňujú ich nevýhody.

Prechádzame poľom

- dva spôsoby prechádzania poľom

```
String[] mená = { "Rex", "Brok" };
```

- klasický

```
for (int i = 0; i < mená.length; i++) {
    String meno = mená[i];
    System.out.println(meno);
}
```

- Java 5:

```
for (String meno : mená) {
    System.out.println(meno);
}
```

nevieme zistiť index prvku, ale často ho netreba

Prechádzame zoznamom

- dva spôsoby prechádzania zoznamom
- klasický

```
for (int i = 0; i < zoznamMien.size(); i++) {
    String meno = zoznamMien.get(i);
    System.out.println(meno);
}
```

- Java 5:

```
for (String meno : zoznamMien) {
    System.out.println(meno);
}
```

Zoznamy primitívov

- zoznam nemôže obsahovať primitívne typy

```
ArrayList<int> čísla = new ArrayList<int>();
```

- každý primitívny typ má svojho „kamaráta" objektového typu
- zoznam môže obsahovať len objektové typy

int	Integer
boolean	Boolean
char	Character
...	...

- našťastie máme **autoboxing** – automatický prevod medzi primitívnymi a objektovými typmi!

Autoboxing inde

- za dávnych čias Javy 1.4 (pred 2005)

```
ArrayList zoznam = new ArrayList();
zoznam.add(new Integer(5));
Integer prvýPrvok = (Integer) zoznam.get(5);
int prvýPrvokAkoInt = prvýPrvok.intValue();
```

- ale prišiel Tiger (Java 5) a zmenil to

```
ArrayList<Integer> zoznam = new ArrayList<Integer>();
zoznam.add(5);

int prvýPrvokAkoInt = zoznam.get(0);
```

autoboxing!: literál 5 typu ***int*** sa automaticky zmenil na objekt typu ***Integer!***

autoboxing!: objekt v zozname typu *Integer* sa automaticky zmenil na primitív typu ***int***

Autoboxing inde

- autoboxing funguje nezávisle od zoznamov

```
Integer päť = 5;
```

autoboxing!: literál 5 typu ***int*** sa automaticky zmenil na objekt typu ***Integer!***

```
byte päťAkoByte = päť.byteValue();
int desať = päť + 5;
```

autoboxing!: objekt typu *Integer* sa automaticky zmenil na primitív typu ***int***, aby ho bolo možné pripočítať k inému primitívu

Ďalšie finty pre prácu so zoznamami

- rýchle vytvorenie nemenného zoznamu

```
List<String> beatles
  = Arrays.asList("Ringo", "John", "Paul", "George");
List<Integer> prvýŤah = Arrays.asList(1, 2, 3, 4, 5);
```

- ***List*** je interfejs pre ArrayList
  - o nich neskôr
  - zatiaľ stačí vedieť, že: List poskytuje rovnaké metódy ako ArrayList

Ďalšie finty pre prácu so zoznamami

- pridanie viacerých prvkov

```
ArrayList<String> goons = new ArrayList<String>();
Collections.addAll(goons, "Peter", "Spike", "Harry");
```

- utriedenie prvkov

```
Collections.sort(goons);
```

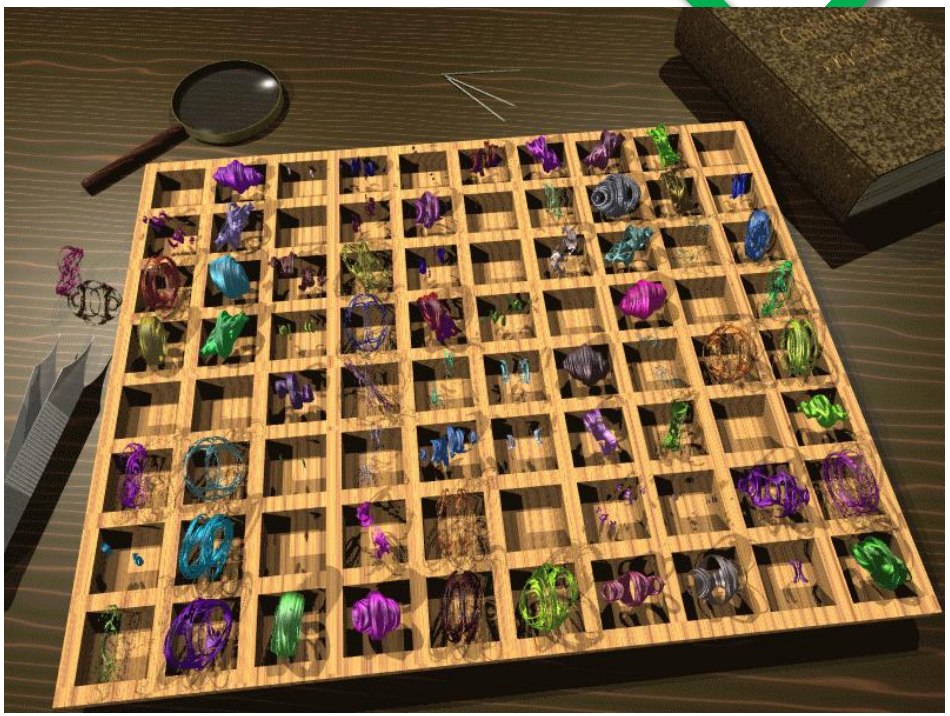
  - interná implementácia používa QuickSort

- maximá a minimá

```
int maximum = Collections.max(zoznamČísiel);
String prvýPodľaAbecedy = Collections.min(goons);
```

Sumár, kde sme

- vieme navrhovať triedy
  - s primitívnymi inštančnými premennými
- vieme používať kolekcie

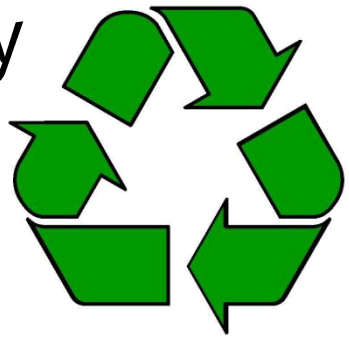


Ako nevynájsť koleso nanovo

- znovupoužiteľnosť kódu je jedným z cieľov OOP
  - nebudeme vynachádzať ni teplú vodu, ni koleso, ni triedenie, ni spojový zoznam...
- dobre navrhnutú triedu možno prevziať a použiť v iných projektoch bez zmien
- vznikajú tak knižnice tried, kde megaprojekt vystaviame zo znovupoužiteľných súčastí

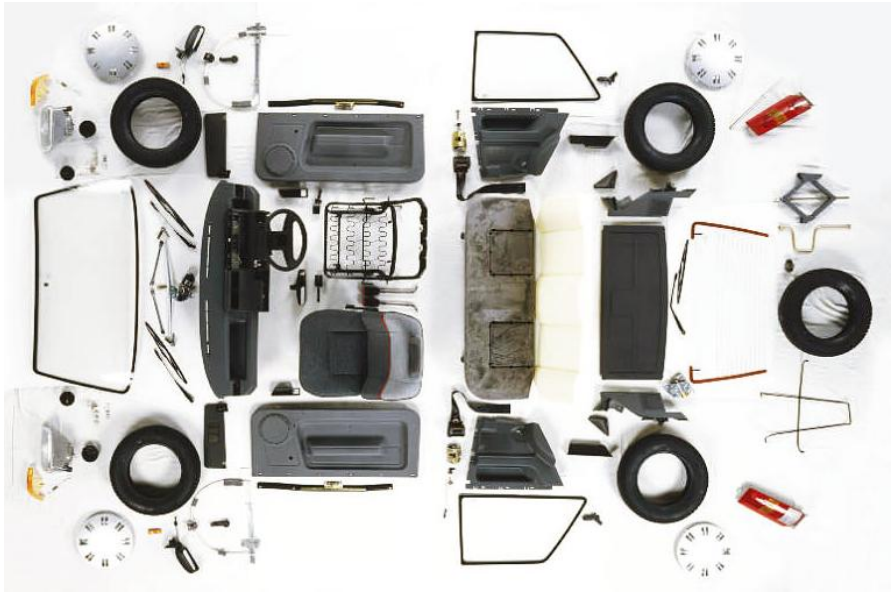
Ako vystavať babylonskú vežu

- v OOP jestvujú dva spôsoby ako vystavať zložitú triedu z jednoduchších
  - kompozícia: skladanie. Stav zložitej triedy je tvorený jednoduchšími triedami
  - dedičnosť: odvodzovanie/špecializácia z jednoduchšej/všeobecnejšej triedy
  - delegácia: hybrid



Kompozícia tried

- Automobil sa skladá z motora, prevodovky, volantu...
  - Motor pozostáva z valcov, sviečok, ...
    - Valec pozostáva z ...



Kompozícia v OOP

Kompozícia v OOP je jednoduchá. Súčasti uvedieme ako stav triedy:

```
class Auto {
   Motor motor;
   Prevodovka prevodovka;
   Volant volant;
}
```

```
public class Motor {
  Valec[] valce;
  Sviečky[] sviečky
}
```

```
public class Valec {
   ...
}
```

V ľudskej reči je kompozícia vyjadrená slovom „má“ (*has a*). Auto má motor. Motor má valec.

Kompozícia v OOP

Inštanciu vystaviame z iných inštancií:

```
class Osoba {
  String meno;
  Adresa bydlisko;
}
```

```
public class Adresa {
  String ulica;
  int číslo_domu;
  String mesto;
  int psč;
}
```

```
Adresa adresa = new Adresa();
adresa.ulica = "Hlavná";
adresa.číslo_domu = 35;
adresa.mesto = "Košice";
adresa.psč = 04001;

Osoba osoba = new Osoba();
osoba.meno = "František Zlý";
osoba.bydlisko = adresa;
```

Chráňte životné prostredie a inštančné premenné!

```
Pes slayer = new Pes();
slayer.vek = -125;
```

Riešenie:

- ku všetkým inštančným premenným budeme pristupovať pomocou metód
- tešíme sa na kopu klepkania!

Chráňte životné prostredie a inštančné premenné!

```
Pes slayer = new Pes();
slayer.nastavVek(25);
int vekPsa = slayer.dajVek();
System.out.println(vekPsa);
```

```
public class Pes {

   int vek;

   void nastavVek(int novýVek) {
       vek = novýVek;
   }

   int dajVek() {
       return vek;
   }
}
```

Chráňte životné prostredie a inštančné premenné!

```
public class Pes {
  //...
  void nastavVek(int novýVek) {
    if(novýVek >= 0) {
      vek = novýVek;
    } else {
      System.out.println("Vek nesmie byť < 0");
    }
  }
}
```

PAZ1C

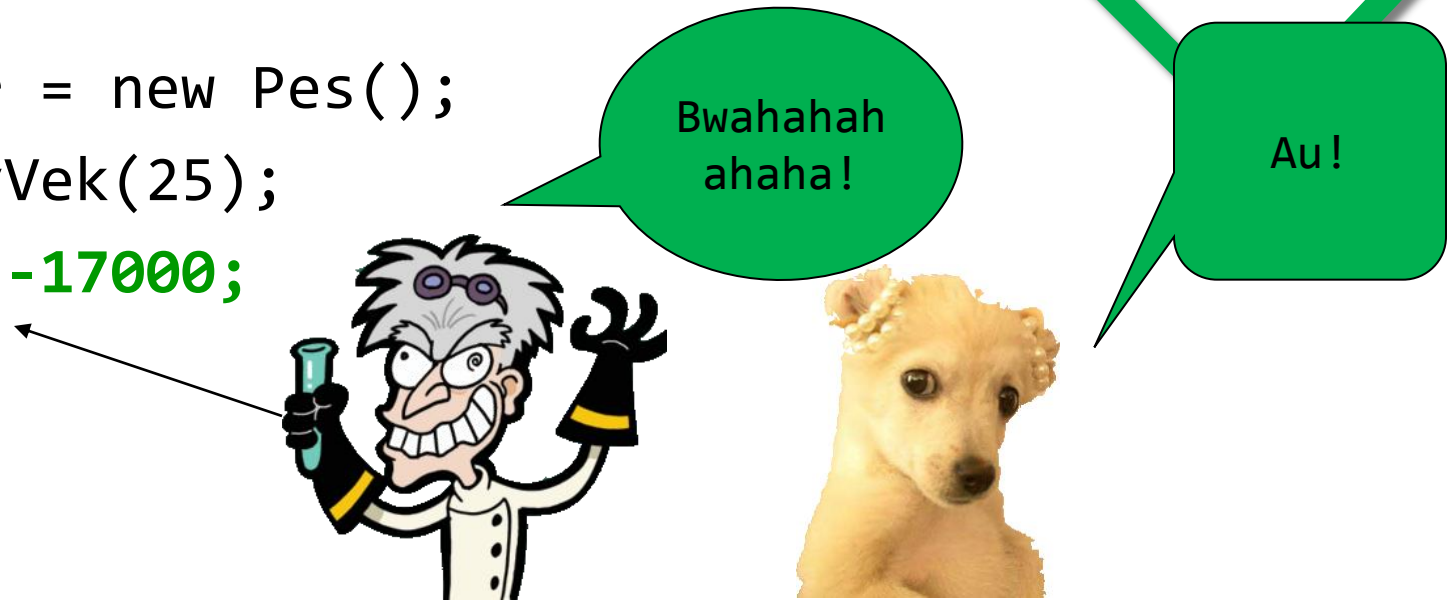
To je lepšie

```
slayer.nastavVek(3);
System.out.println(slayer.dajVek());
slayer.nastavVek(-125);
System.out.println(slayer.dajVek());
```

```
3
Vek nesmie byť < 0
3
```

Chráňte životné prostredie a inštančné premenné!

```
Pes slayer = new Pes();
pes.nastavVek(25);
pes.vek = -17000;
```



Ako zabrániť zlému vedcovi páchať neprístojnosti?

Vyhlásime inštančnú premennú v triede `Pes` za *súkromnú*

```
private int vek;
```

Súkromné vlastníctvo

- k privátnym premenným môže pristupovať len kód v danej triede

```
public class Pes {
   private int vek;

   int dajVek()
   {
        return vek;
   }
   void nastavVek(int novýVek) {
        vek = novýVek;
   }
}
```

```
Pes slayer = new Pes();

slayer.vek = -17000;
```

Súkromné vlastníctvo

- Príklad použitia: chceme premennú, do ktorej nemôžeme zapisovať („read only")

```
public class Pes {
   private int vek;

   int dajVek() {
       return vek;
   }

}
```

```
Pes slayer = new Pes();
slayer.vek = -17000;
slayer.nastavVek(25);
```

Taká premenná neexistuje!

Taká metóda neexistuje!

Daj a nastav po našom

- Pre každú inštančnú premennú naklepkáme dve metódy:
  - `dajXXX` `dajVek()`, `dajRasu()`
  - `nastavXXX` `nastavVek(...)`, `nastavRasu()`
- Premenné určené len na čítanie majú len jednu metódu
  - `dajXXX`

Daj a nastav po americky

- **Dohoda:**
  - metóda `dajXXX()` sa bude zapisovať ako `getXXX()`
  - metóda `nastavXXX()` sa bude zapisovať ako `setXXX()`

„nahodíme gettre a settre"

Nariadenie

Odteraz pristupujeme k inštančným premenným triedy len cez `getXXX()` a `setXXX()`!

Daj a nastav po americky

- Problém so slovenčinou
  - správne by malo byť: `String dajRasu()`
  - podľa dohody: `getRasu()`
- **zabudneme** na Ľ. Štúra a na skloňovanie
  - nástroje predpokladajú takýto stav:
    - `private String rasa;`
    - get + rasa = **`getRasa()`**
    - set + rasa = **`setRasa(String novaRasa)`**
- **zabudneme na slovenčinu** = riešenie odrodilca

„reč, ktorú z domu vieš, ó, jak je lichá"

```
private String breed;
String getBreed()
setBreed(String aBreed)
```

Gettre a settre

- prístup k všetkým inštančným premenným cez metódy – **úplné zapúzdrenie**
- metódy našťastie nemusíme klepkať!
  - pomôže nám IDE
  - **Eclipse**: Source | Generate getters and setters...