Programovanie, algoritmy, zložitosť (UINF / PAZ1c)

Diel II.

Róbert Novotný
robert.novotny@upjs.sk
1. 10. 2008

V minulom dieli ste videli...

```
class Pes {
  String rasa;

  int vek;

  String stekaj() {
     return "Haf!"
  }

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

inštančné premenné

metódy

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!

```
public class Pes {

   //.. rasu vynecháme

   int vek;


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


   int dajVek() {
        return vek;
   }

}
```

```
Pes slayer = new Pes();

slayer.nastavVek(25);

int vekPsa = slayer.dajVek();

System.out.println(vekPsa);
```

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");
       }
   }
}
```

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*

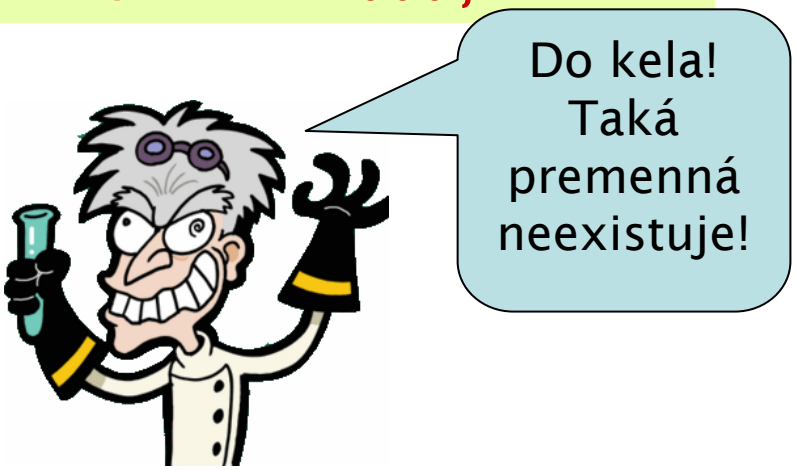
```
private int vek;
```

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 americky

- Pre každú inštančnú premennú naklepkáme 2 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 premenným triedy len cez `getXXX` a `setXXX`

„*úplné zapúzdrenie*"

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 novárasa)`
- zabudneme na slovenčinu = riešenie odrodilca

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

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

Preťažené metódy (nie výťahy)

- Niekedy je vhodné mať metódy s rovnakým názvom
- Preťažené (*overloaded*) metódy
- Ak majú dve metódy rovnaký názov, **musia** mať rôzne typy parametrov.

```
void setVáha(int nováVáha) {
   váha = nováVáha;
}
void setVáha(float nováVáha) {
   //prevedieme desatinný int na celé číslo
   //odrežeme des. miesta
   váha = new Float(nováVáha).intValue();
}
```

dve metódy s rovnakým názvom

Preťažené metódy (nie výťahy)

- Príklad: notoricky známa metóda `println()` v `System.out`
- 9 preťažených metód: pre každý primitívny dátový typ jedna
- Ak by neexistovalo preťaženie, museli by sme mať metódy
  - `printlnInt(int i);`
  - `printlnFloat(float f);`
  - `....`

Preťažené metódy (nie výťahy)

- Pozor: metódy s rôznymi návratovými typmi sa nepovažujú za preťažené, ba priam sa nie sú povolené

```
public class Pes {
        String štekaj() {
                return "Haf haf";
        }
        int štekaj() {
                return 0101010101;
        }
}
```

```
Pes dunčo = new Pes();
dunčo.štekaj();
```

Konštruktory

- Spomnime:

```
Pes dunčo = new Pes();
```

- Načo sú tam tie dve zátvorky?
- Odpoveď:
  - existuje špeciálna metóda, ktorá sa zavolá pri vytváraní objektu na halde
  - zvaná ***konštruktor***
  - užitočná pri úvodnom nastavení objektu
- Kde boli konštruktory doteraz?
  - boli neviditeľné, resp. prázdne

Vypíšte správu pri zrodení nového psa

- Riešenie: v triede `Pes` vytvoríme nový konštruktor

```
public class Pes {
	Pes() {
		System.out.println("Haf!");
	}
}
```

```
Pes lajka = new Pes();
Pes dunčo = new Pes();
```

```
Haf
Haf
```

Konštruktory

```
public class Pes {

      Pes() {
            System.out.println("Haf!");
      }
}
```

- názov konštruktoru **musí byť rovnaký** ako názov triedy
- konštruktor **nemá** uvedený návratový typ a v jeho tele sa **nevracia** žiadna hodnota

```
public class Pes {
  void Pes() {
        ...
   }
}
```

```
public class Pes {
  Pes Pes() {
        return this;
   }
}
```

Konštruktory s parametrami

```
public class Pes {
        ...
        Pes(String nováRasa, int novýVek) {
                rasa = nováRasa;
                vek = novýVek;
        }
}
```

- konštruktory môžu mať parametre
- presne ako metódy

```
Pes dunčo = new Pes("buldog", 25);
System.out.println(dunčo.getVek());
```

Parametrami preťažení konštruktéri

- konštruktorov môžeme mať koľko len chceme

```
public class Pes {
        ...
        Pes(String nováRasa, int novýVek) {
                rasa = nováRasa;
                vek = novýVek;
        }
        Pes(String nováRasa) {
                rasa = nováRasa;
        }

        Pes() {
                //nerob nič
        }

}
```

Parametrami preťažení konštruktéri

- Pozor! Ak používame preťažené konštruktory a chceme používať aj prázdny konštruktor, musíme ho v triede explicitne napísať!

```
public class Pes {
        Pes(String nováRasa, int novýVek) {
                rasa = nováRasa;
                vek = novýVek;
        }
}
```

```
Pes dunčo = new Pes();
```

The constructor Pes() is undefined!
/
Konštruktor Pes() nie je definovaný!

Parametrami preťažení konštruktéri

- Pozor! Ak používame preťažené konštruktory a chceme používať aj prázdny konštruktor, musíme ho v triede explicitne napísať!

```
public class Pes {
  Pes() {
    // prázdny konštruktor
  }

  Pes(String nováRasa, int novýVek) {
    rasa = nováRasa;
    vek = novýVek;
  }
}
```

Dátové štruktúry

- s obyčajnými dátovými typmi by sme sa ďaleko nedostali
- potrebujeme často reprezentovať komplikované štruktúry
- spomni matematiku:
  - množiny

„Teória množín je programovací jazyk.“

  - zoznamy
  - funkcie (postupnosti)
  - stromy (grafy)

Polia obyčajné

„...a od Prešova v tym poľu"

- už poznáme
- deklarácia a inicializácia

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

pole dĺžky 4

- alternatívna deklarácia a inicializácia

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

pole dĺžky 2

- prístup k prvkom poľa

```
String prvý = reťazce[0];
```

- dĺžka poľa

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

v kučeravých zátvorkách vymenujeme prvky poľa

bez zátvoriek! toto nie je metóda

Výhody a nevýhody polí

+ dĺžka poľa môže byť určená premennou
- 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();
```

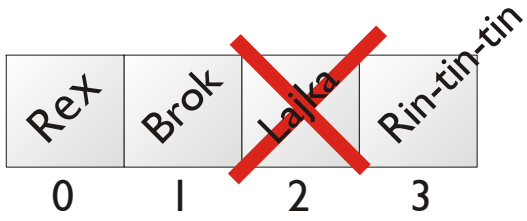
so zátvorkami!
toto je metóda!

Polia dynamické čili zoznamy

- odstránenie prvku

```
mená.remove(2);
```

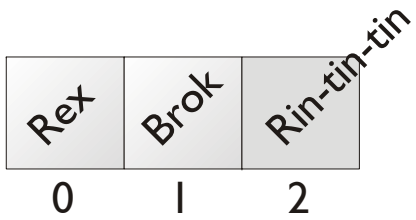
  - prvky sa posunú doľava



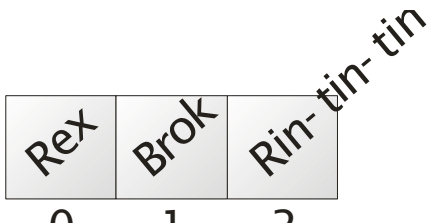
- pridanie prvku na pozíciu

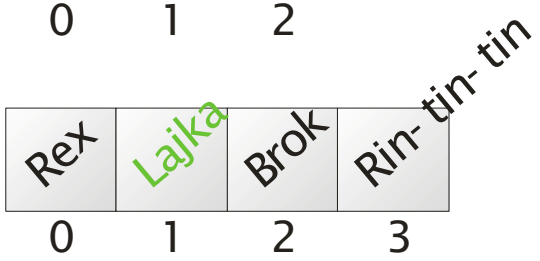
```
mená.add(1, "Lassie");
```

  - pridanie na ,,druhú" pozíciu
  - prvky sa posunú doprava



- ostatné operácie
  - viď dokumentácia

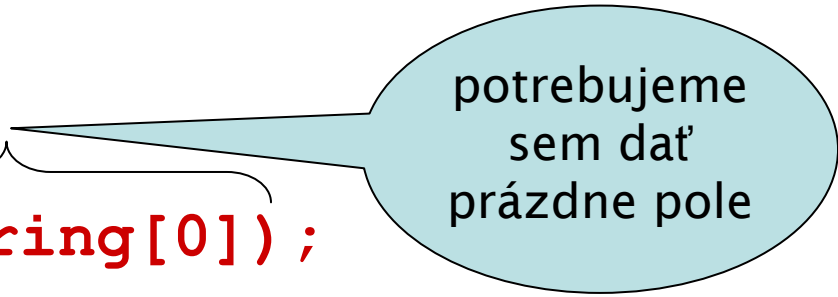




Polia vs. zoznamy

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

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



- pole => zoznam

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

- porovnanie
  - prístup k poliam je rýchlejší – chlieviky v pamäti za sebou
  - v Jave 5: 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);
}
```

Ďalšie dátové štruktúry

- v starej Jave – namiesto ArrayList-ov bol `java.util.Vector`.
  - radšej používajme `ArrayList`
- existujú aj ďalšie dátové štruktúry
  - množina – `java.util.HashSet`
  - mapa – `java.util.HashMap`
  - o nich neskôr...

Utajené metódy práce s kávou

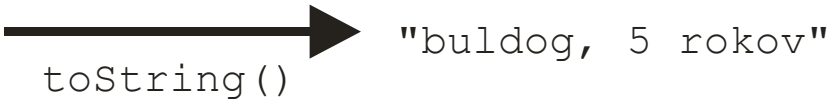
- Je užitočné vedieť o niektorých ,,utajených metódach" práce s objektami
- traja zhavranelí bratia
  - `toString()`

    ,,znak si a v znak sa obrátiš"
  - `equals()`

    ,,všetci sú si rovní, len niektorí sú si rovnejší"
  - `hashCode()`

    ,,heš! heš!"

Metóda `toString()`

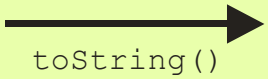


- Každá trieda môže mať metódu s hlavičkou

```
public String toString()
```

- Metóda slúži na vrátenie ľubovoľného **reťazca**, ktorý popisuje objekt
- Často používaná na ladenie

Metóda `toString()`




```
class Pes {
   private String rasa;
   private int vek;
   …
   public String toString() {
        String s = "Pes rasy " + rasa + " s vekom " + vek;
        return s;
   }
}
```

```
Pes lajka = new Pes();
lajka.setRasa("buldog");
lajka.setVek(5);
System.out.println(lajka.toString());
```

```
Pes rasy buldog s vekom 5
```

```
System.out.println(lajka);
```

pri volaní metódy `println()` sa metóda `toString()` volá automaticky

Metóda `toString()`

```
public String toString() {
   return String.format("Pes rasy %s s vekom %d", rasa, vek);
}
```

- Metóda `format` funguje na spôsob `printf()` v Cčku
- Užitočná na formátovanie výstupu
  - **`%s`** a **`%d`** sú zástupné znaky
  - **`%s`** je zástupný znak pre reťazec. V príklade sa nahradí hodnotou premennej rasa
  - **`%d`** je zástupný znak pre celé číslo. V príklade nahradí hodnotou premennej vek

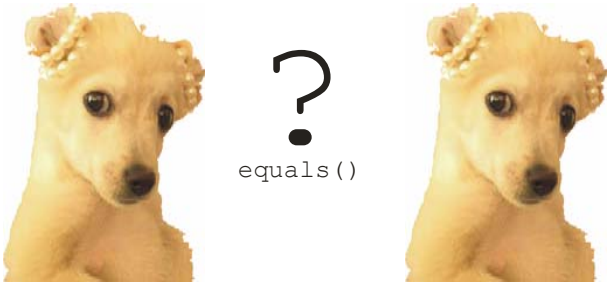
Formátovanie ~~disku~~ reťazcov

```
String prvy = "Miško";
String druhy = "Ferko";
int vek1 = 25;
float vek2 = 35.0f;

String.format("%s - %d, %s - %f", prvy, vek1, druhy, vek2)
```

- Prvý parameter je reťazcová premenná `prvy`,
- Druhý celé číslo `vek1`
- Tretím reťazec druhy
- Štvrtým reálne číslo `vek2`

```
Miško - 25, Ferko - 35.0
```

Metóda `equals()`



- Slúži na **porovnávanie** objektov
- Ale ako môžeme porovnávať dva objekty?
  - databázovo
    - ak majú dva objekty rovnaké ID, sú rovnaké
    - čo dvaja občania s rovnakým rodným číslom?
  - porovnaním atribútov
    - dva psy sú rovnaké, ak majú rovnakú rasu a rovnaký vek

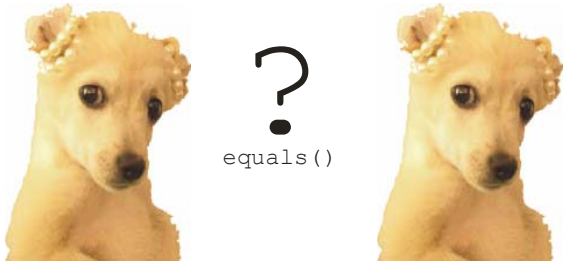
„veľmi zjednodušené"

- Každá trieda môže mať metódu

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

Metóda `equals()`

- Častejšie sa zvykne používať porovnávanie atribútov
- Metóda sa ľahko pochopí, ale ťažko napíše




```
public boolean equals(Object o) {
   Pes pes = (Pes) o;

   if(pes.getRasa().equals(rasa)
      && pes.getVek() == vek)
   {
      return true;
   else {
      return false;
   }
}
```

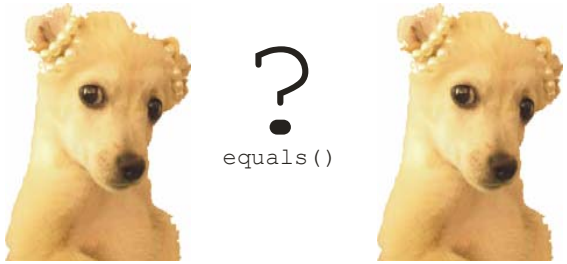
Keďže parameter je `Object` (teda hocičo), musíme zmeniť dátový typ zo všeobecného `Object`-u na `Psa`

(Viď dedičnosť)

paz1c

Metóda `equals()`

- Častejšie sa zvykne používať porovnávanie atribútov
- Metóda sa ľahko pochopí, ale ťažko napíše



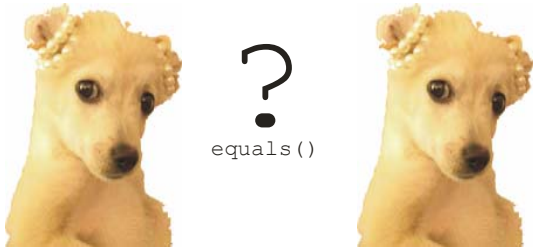

```
public boolean equals(Object o) {
   Pes pes = (Pes) o;

   if(pes.getRasa().equals(ras
      && pes.getVek() == vek)
   {
      return true;
   else {
      return false;
   }
}
```

Tu môže prísť hocičo, aj objekt typu `Asteroid`

Nesmierne komplikácie pri dedičnosti, treba ošetriť veľa bočných prípadov

Metóda `equals()`

- Metóda sa ľahko pochopí, ale ťažko napíše



Čítanie z listov JavaDocu, kapitola `Object`, metóda `equals()`;

Vlastnosť ,,rovná sa" musí byť

- reflexívna
- symetrická
- tranzitívna
- konzistentná
- negatívna vzhľadom na `null`

Metóda `equals()`

- Riešenie: inžiniersky prístup: použitím IDE (Eclipse) a jeho nástroja na vygenerovanie metódy.

```
public boolean equals(Object obj) {
   if (this == obj) return true;
   if (obj == null) return false;
   if (getClass() != obj.getClass()) return false;
   final Pes other = (Pes) obj;
   if (rasa == null) {
        if (other.rasa != null)
            return false;
   } else if (!rasa.equals(other.rasa))
        return false;
   if (vek != other.vek)
        return false;
   return true;
}
```

Metóda `hashCode()`

ak to nie je splnené, množiny nebudú fungovať

- Používaný v algoritmoch triedy pre množiny HashSet
- Každý objekt by mal mať svoj čo najjedinečnejší kód.
- Požiadavky:
  - ak sú dva objekty rovnaké (`equal`), majú rovnaký `hashCode`
  - dva nerovnaké objekty môžu mať rovnaký hashCode
- Opäť inžiniersky prístup: generovanie pomocou IDE