Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

Róbert Novotný
robert.novotny@upjs.sk

30. 9. 2009

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.

dve metódy s rovnakým názvom

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

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

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

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

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

PAZ1C

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

- názov konštruktora **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

PAZ1C

- konštruktorov môžeme mať koľko len chceme

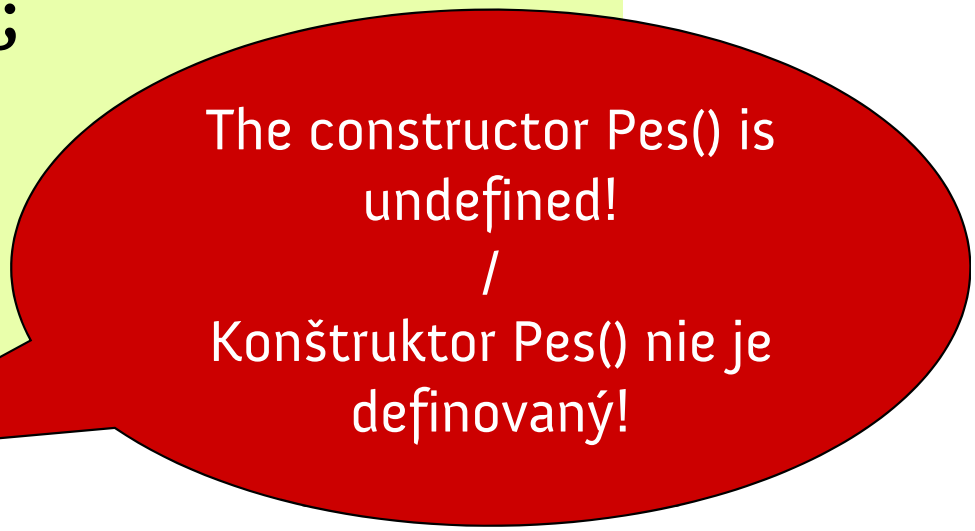
```
public class Pes {
        ...
        Pes(String nováRasa, int novýVek) {
                rasa = nováRasa;
                vek = novýVek;
        }
        Pes(String nováRasa) {
                rasa = nováRasa;
        }
        Pes() {
                //nerob nič
        }
}
```

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();
```

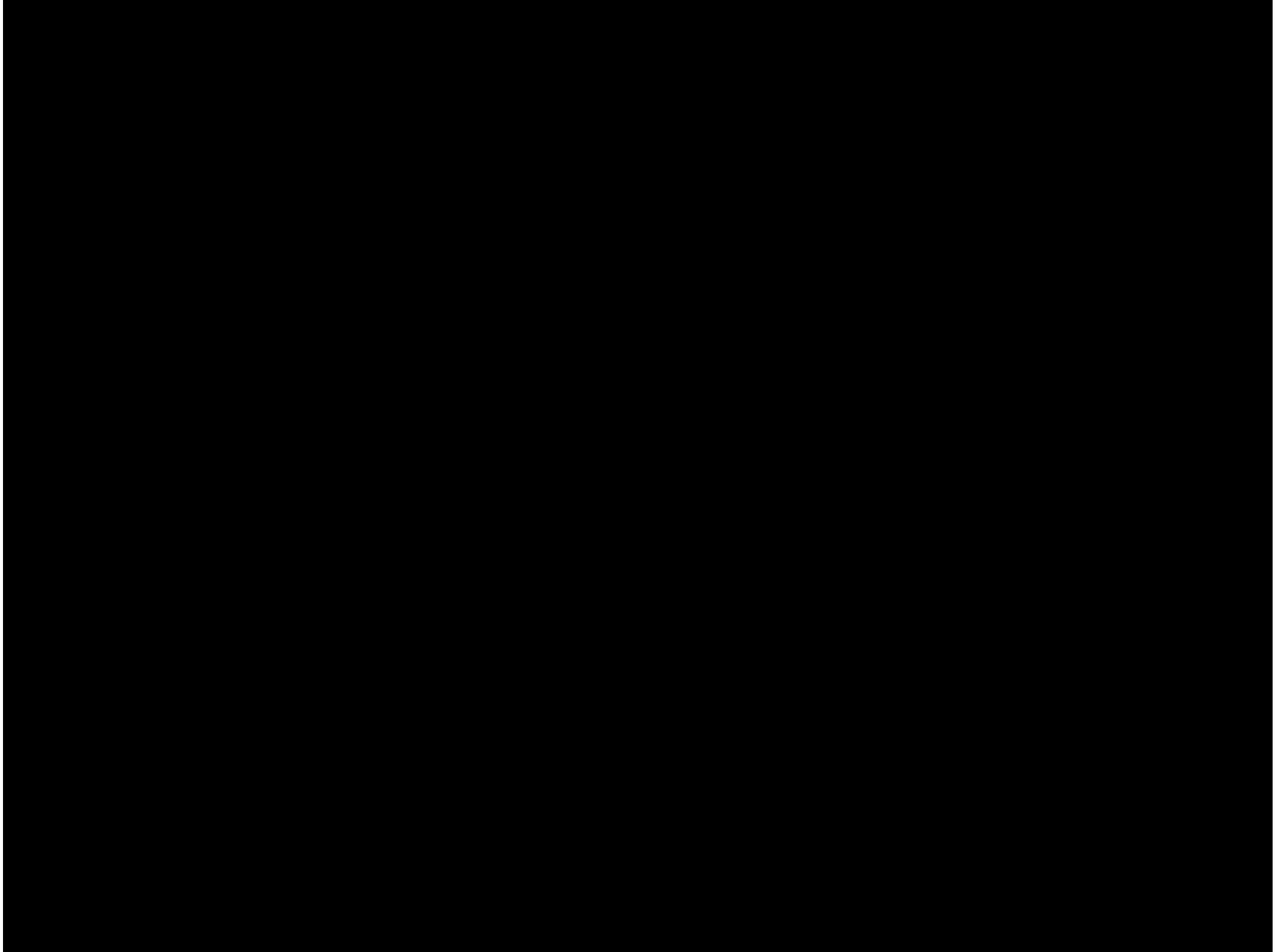


Parametrami preťažení konštruktéri

- 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;
  }
}
```



Zapúzdrenie

- *„proces „zaškatuľkovania" elementov abstrakcie, ktoré tvoria jej štruktúru a správanie. Účelom zapúzdrenia je oddeliť rozhranie s kontraktom abstrakcie od jej implementácie"*

—Grady Booch, *Object-Oriented Analysis and Design with Applications*, 2007

Štruktúra a správanie

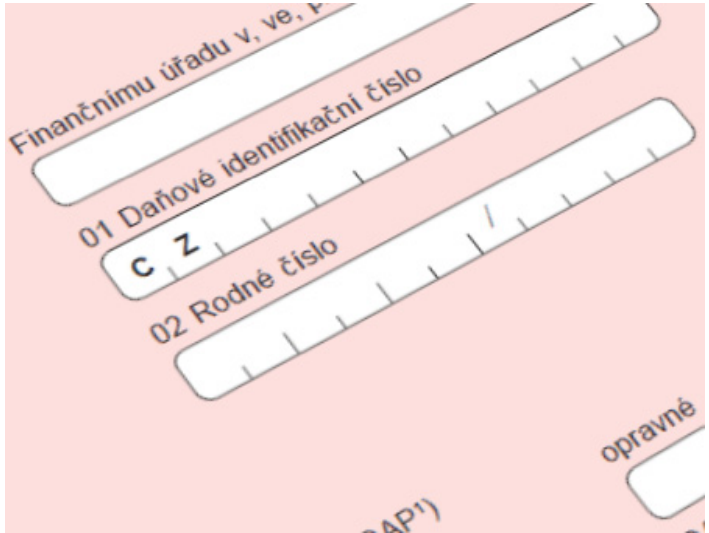
- *štruktúra* = **stav** = inštančné premenné
- *správanie* = **schopnosti** = metódy
- *„zaškatuľkovaný" element abstrakcie* = **trieda**
- *kontrakt* = **hlavičky metód** = formálna syntax pre to, čo od triedy očakávame
- *implementácia* = **kód** v metódach

Stav a schopnosti

- aký **stav** a **schopnosti** má mať trieda?
- úvahy nad kontraktom by mali mať prednosť pred úvahami nad implementáciou
- návrh stavu často záleží od očakávaných schopností

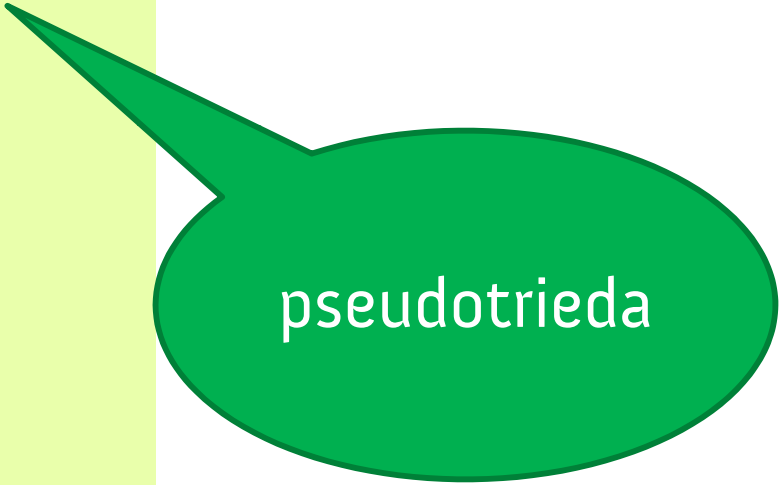
Príklad s rodným číslom

- aký **stav** a **schopnosti** má mať RČ?
- pýtajme sa naopak: najprv zistime schopnosti, od nich odvoďme stav
- aké **schopnosti** očakávame od rodného čísla?
  - zistiť deň, mesiac a rok narodenia
  - zisti, či je korektné
  - zisti, či je majiteľ muž alebo žena
  - daj reťazcovú reprezentáciu
    - s lomkou i bez



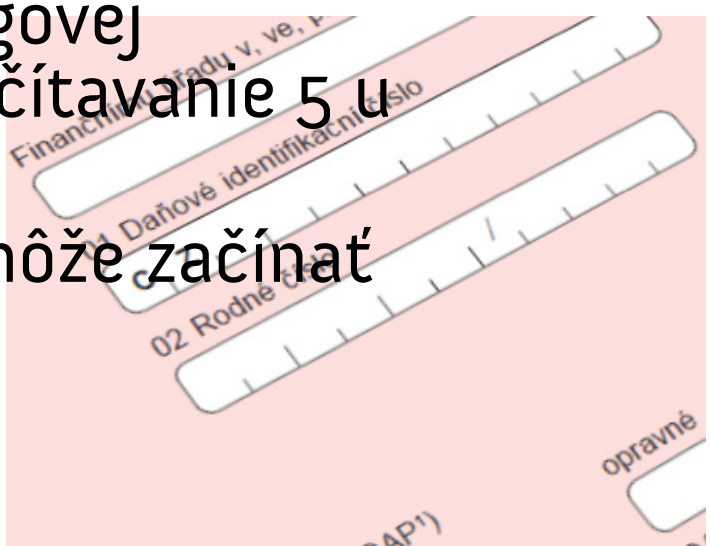
Príklad s rodným číslom


```
class RodneCislo {
  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```



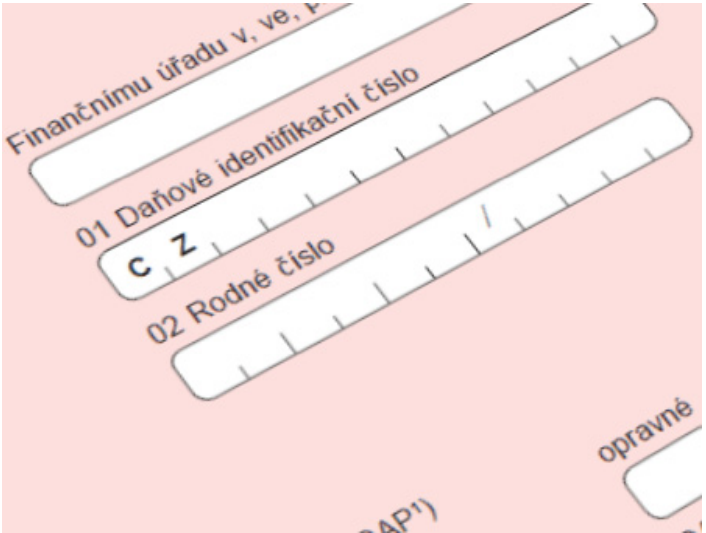
Rodné číslo – ako reprezentovať?

- ako interne reprezentovať rodné číslo?
- možnosť 1: jeden String
  - nevýhoda: ak chceme vypisovať s lomkou / bez lomky, musíme manipulovať so Stringami
  - nevýhoda: validácia = deliteľnosť štyroma
- možnosť 2: štyri integerové premenné: deň, mesiac, rok, prípona
  - nevýhoda: treba vysekávať zo Stringovej reprezentácie, nezabudnúť na pripočítavanie 5 u žien
  - nevýhoda: pozor na to, že prípona môže začínať nulou, to isté jednotlivé zložky



Rodné číslo – ako reprezentovať?

- ako interne reprezentovať rodné číslo?
- možnosť 3: jeden int
  - nevýhoda: treba vysekávať zo Stringovej reprezentácie, nezabudnúť na pripočítavanie 5 u žien
  - pozor na to, že rodné číslo môže začínať nulou (deti narodené 2000-2009)



Akú reprezentáciu zvoliť?

- Používateľa triedy nezaujíma, akú reprezentáciu zvolíme, dôležité je, že jeho metódy robia to, čo sa od nich čaká
- vonkajší pohľad na triedu ustanovuje **kontrakt**
  - záruky správania a predpoklady na ktoré sa môže používateľ triedy spoliehať
  - očakávania používateľa, ktoré musí trieda naplniť
  - kontrakt zároveň určuje zodpovednosti triedy

Akú reprezentáciu zvoliť?

- **Kontrakt** je definovaný hlavičkami verejných metód
  - parametre a ich typy
  - návratové hodnoty
- Správanie metód je záležitosťou implementácie
  - trieda sa správa ako čierna skrinka

Príklad s rodným číslom

- v triede RodnéhoČísla sme definovali kontrakt
- implementácia spočíva v návrhu inštančných premenných a v interných algoritmoch, ktoré využívajú stav

```
class RodneCislo {
  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```

Príklad s rodným číslom

- zvoľme si reprezentáciu jedným Stringom
- jedna inštančná premenná
- kód v metódach bude závisieť na reprezentácii
- `jeMuzske()` – zistí, či sa 5. miesto začína nulou alebo jednotkou
- ...

```
class RodneCislo {
  private String rč;
  RodneCislo(String rc);

  int getRok()
  int getMesiac()
  int getDen()
  boolean jeMuzske()
  boolean jeValidne()
  String toString()
  String toStringBezLomky()
}
```

Príklad s rodným číslom

- Použitie triedy:

```
RodnéČíslo rč = new RodnéČíslo("751212/8823");
if(rč.jeValidné()) {
  System.out.println("Osoba sa narodila"
    + " v roku " + rč.getRok());
}
```

Zapúzdrenie a jeho výhody

- Vďaka zapúzdreniu môžeme v prípade potreby zmeniť internú implementáciu
- Ak dodržíme kontrakt, používateľ si nič nevšimne
- Čierna skrinka!

Príklad s rodným číslom

- Zvoľme štyri integerové premenné:

```
class RodneCislo {
  private int deň;
  private int mesiac;
  private int rok;
  private int prípona;
  ...
}
```

- Ak dodržíme kontrakt, použitie kódu je rovnaké.

```
RodnéČíslo rč = new RodnéČíslo("751212/8823");
if(rč.jeValidné()) {
  System.out.println("Osoba sa narodila"
    + " v roku " + rč.getRok());
}
```


PAZ1C

Reprezentácia dátových typov v pamäti počítača

- alias *„Bratia == a equals() zasahujú“*
- jestvujú dva druhy dátových typov: primitívy a objekty
  - primitívy: `int`, `boolean`, `float`, `double`,...
    - dátové typy začínajúce malým písmenom
  - objekty: `String`, `Pes`,...
    - začínajúce veľkým písmenom

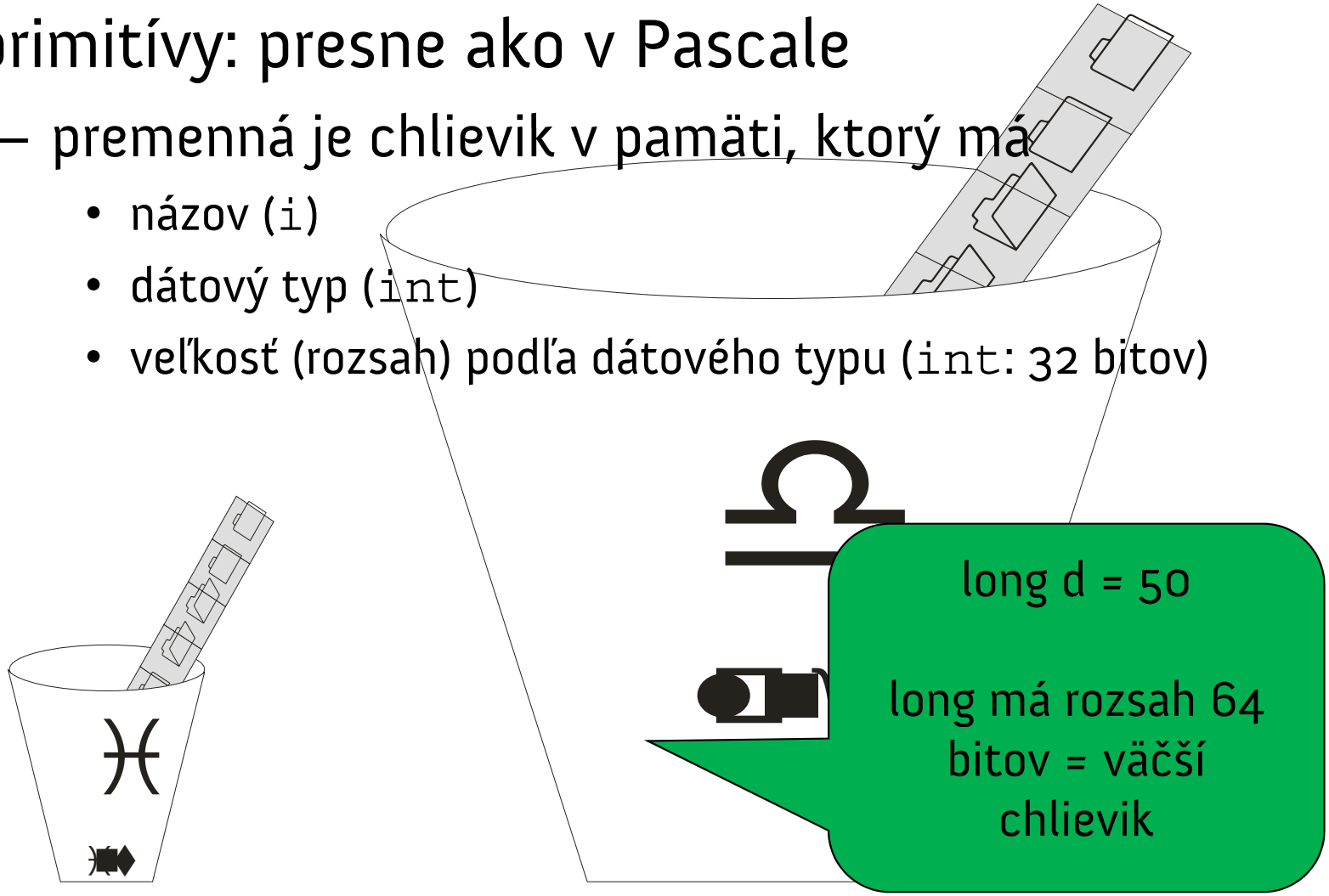
Reprezentácia primitívov v pamäti počítača

- primitívy: presne ako v Pascale
  - premenná je chlievik v pamäti, ktorý má
    - názov (`i`)
    - dátový typ (`int`)
    - veľkosť podľa dátového typu (`int`: 32 bitov)
- príklad: `int i = 52`. V binárnom kóde: `110100`

i je názov tridsiatich dvoch chlievikov v pamäti

Reprezentácia primitívov v pamäti počítača

- primitívy: presne ako v Pascale
  - premenná je chlievik v pamäti, ktorý má
    - názov (`i`)
    - dátový typ (`int`)
    - veľkosť (rozsah) podľa dátového typu (`int`: 32 bitov)



Reprezentácia primitívov v pamäti počítača

dátový typ	*veľkosť*
int	32 bitov
float	32 bitov
boolean	ťažko povedať, povedzme 1 bit
double	64 bitov
byte	8 bitov

- porovnanie primitívov: výhradne cez ==
- porovnajú sa chlieviky bit po bite.
  - 110100 (50) == 110100 (50)
  - 110101 (51) != 110100 (50)

Reprezentácia primitívov v pamäti počítača - objekt

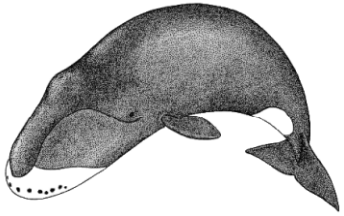
- Objekt je premenná typu špecifikovaného triedou objektu.
- premenná je chlievik v pamäti, ktorý má
  - názov (`i`)
  - dátový typ (`int`)
  - veľkosť podľa dátového typu
- Veľkosť?
  - aký veľký je `String`? A `Pes`? A `Veľryba`?

Je `Veľryba` väčšia než `Mravec`?

- nevieme, aký veľký je objekt
- objekty nemôžeme natlačiť do chlievika
  - čo keď sa nezmestia?
  - *jak dostat velrybu do pohárku*
  - nemôžeme mať nekonečne veľký chlievik

Riešenie

- smerní...ehm, referencie



<

Všetky objekty sú na kope... teda halde

PAZ1C

Urob si haldu v pamäti počítača, v halde urob priehrady a zvnútra i zvonka ich vymaž smolou! A postav ju takto: tristo MB bude jej dĺžka, päťdesiat MB jej šírka a tridsať MB jej výška.

Do korába vojdeš ty i tvoji synovia, tvoja žena aj ženy tvojich synov s tebou.

Zo všetkých vtákov podľa svojho druhu, z dobytka podľa svojho druhu a z plazov podľa svojho druhu vojdú po dvoch do korába s tebou, aby mohli žiť.

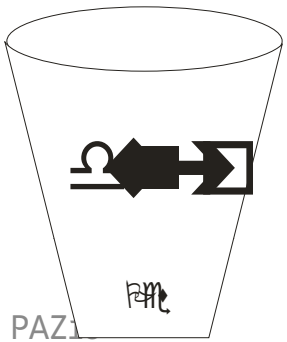
– IT Genesis

Všetky objekty sú na kope... teda halde

- halda (heap) je priestor v pamäti určený pre objekty
  - nemýliť si s heapsortom (triedenie haldovaním)!
- je spravovaný automaticky Javou
- prostý programátor nevie o existencii haldy
- na halde sa dejú kadejaké zverstvá
  - automatické uvoľnovanie pamäte (Garbage Collection)
  - o tom však neskôr

Detaily v útrobách psa

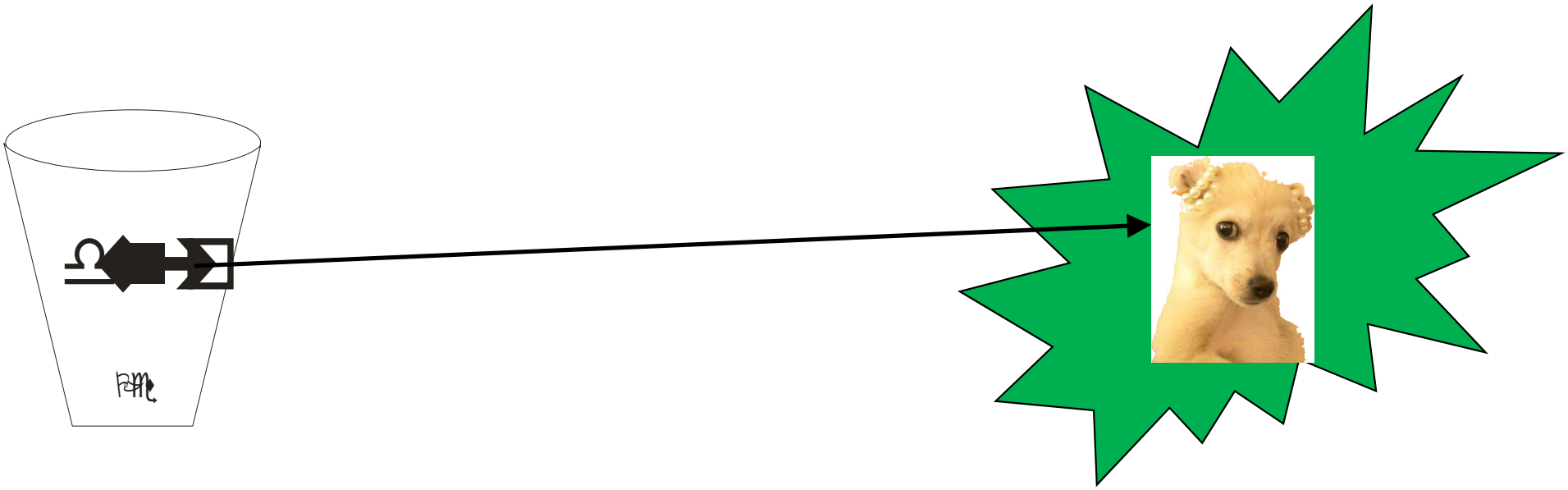
- `Pes dunčo = new Pes()`
- `Pes dunčo;`
  - vytvorí sa nové premenná `dunčo` typu `Pes`
- `new Pes()`
  - na halde sa vytvorí dostatok pamäte pre novú inštanciu



Všetky objekty sú na kope... teda halde

- `Pes dunčo = new Pes()`
- premenná `dunčo` je nasmerovaná na inštanciu psa na halde. Bude obsahovať adresu inštancie na halde

„tretí chlievik zhora, piaty sprava, vedľa veľryby"



Adresa ja, adresa ty...

- premenná `dunčo` obsahuje adresu inštancie na halde
- to je presne idea smerníkov
- našťastie:
  - smerníky sú v pozadí
  - užívateľ ich nevidí
    - ani nechce vidieť
  - žiadne ^ . ako v Pascale

Podobenstvo s diaľkovým ovládaním

- premenná Pes obsahuje „diaľkové ovládanie" inštancie na halde

```
Pes aibo = new Pes();
```

1. **`Pes aibo`** ... vytvoríme novú premennú

   teda pohárik s diaľkovým ovládaním

2. **`new Pes()`** ... vytvorenie novej inštancie na halde

3. **`priradenie`** .. diaľkové ovládanie naprogramujeme na ovládanie konkrétnej inštancie (teda Aiba).

Podobenstvo s diaľkovým ovládaním

- Ak pohárik `int` má veľkosť 32 bitov, akú veľkosť má pohárik typu `Pes`?
- Nevedno, ale ani nás to netrápi.
  - JDK od Sunu: 64 bitov
  - Java od Janka Hraška:
    - 64 bitov na obed
    - 32 bitov v noci

Otázka k diaľkovým ovládaniam

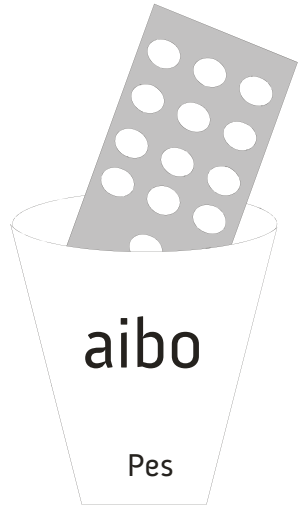
- Ak nadeklarujem premennú a nepriradím jej nič, koho riadi diaľkové ovládanie?

  `Pes aibo;`

  Premenná, ktorej nebola priradená žiadna inštancia ukazuje na **`null`.**

  – **`null`** je analógia **nil** z Pascalu: smerník, ktorý ukazuje nikam.

  Ak premenná ukazuje na null, mám diaľkové ovládanie, ale nemám k nemu televízor.



Dôsledky diaľkového ovládania

PAZ1C

- Čo spraví nasledovný kód?

```
Pes dunčo = new Pes();
dunčo.setRasa("čuvač„);
dunčo.setVek(25);
System.out.println(dunčo.getVek());

Pes aibo = dunčo;
aibo.setVek(35);
System.out.println(aibo.getVek());

System.out.println(dunčo.getVek());
```

```
25
35
```

```
35
```

Ale prrrrečo?


```
Pes dunčo = new Pes();
```

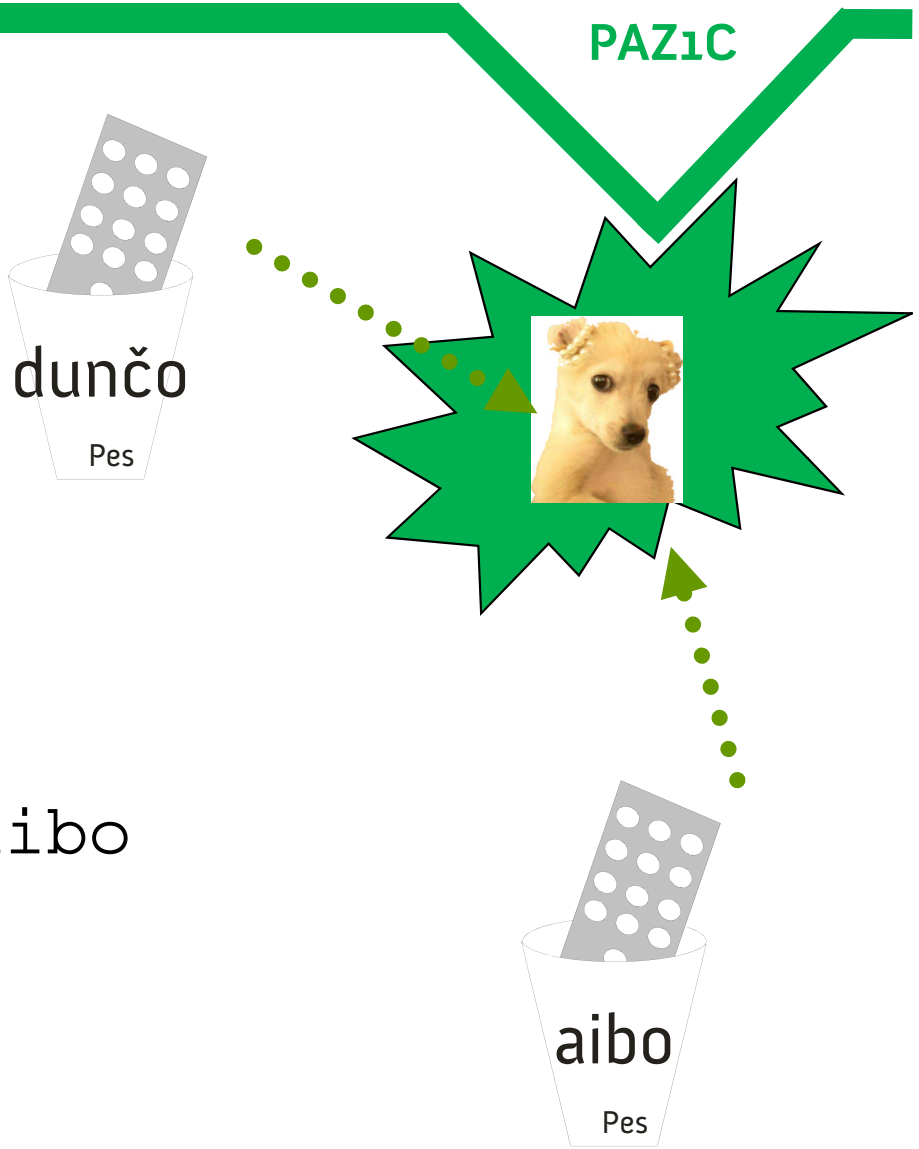
```
Pes aibo  = dunčo
```

- obe diaľkové ovládania riadia toho istého psa
- ak zmeníme vek pomocou *aiba*, zmení sa aj vek pre *dunča*

Dôsledky diaľkového ovládania

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

if(dunčo == aibo) {
    //platí
}
```

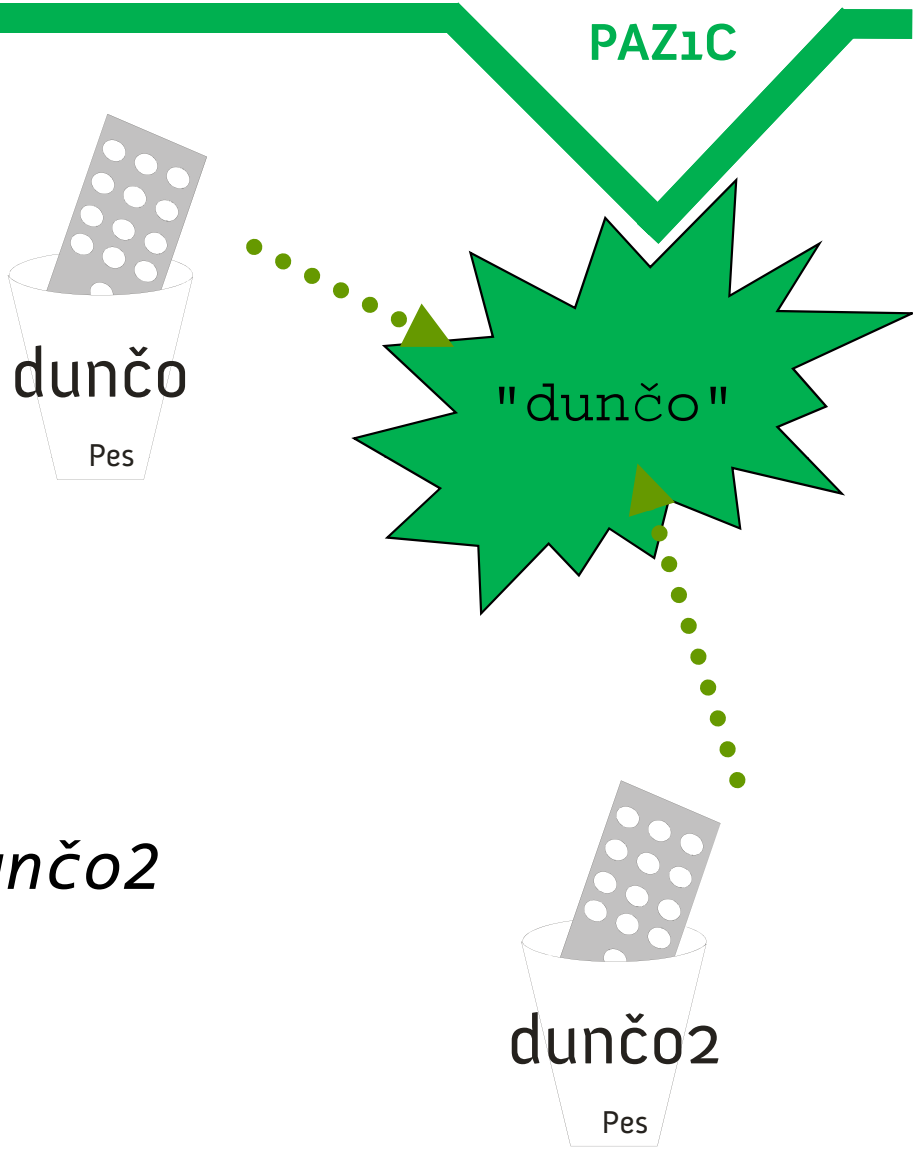


- Podmienka platí, lebo `dunčo` aj `aibo` ukazujú na toho istého psa.
- Toto je však výnimočná situácia.

Dôsledky diaľkového ovládania

```
String dunčo = "dunčo";
String dunčo2 = dunčo;

if(dunčo == dunčo2) {
    //platí
}
```

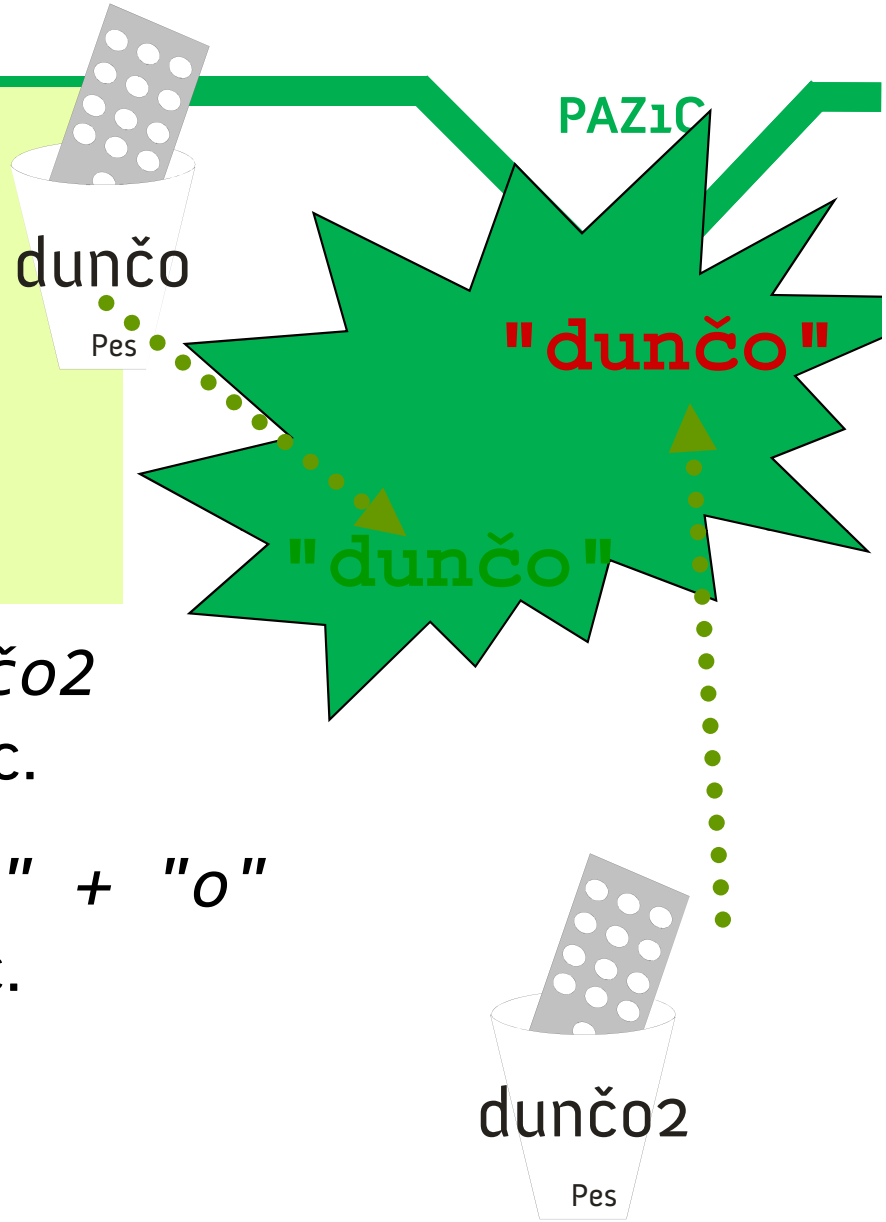


- Podmienka platí, lebo *dunčo* aj *dunčo2* ukazujú na ten istý reťazec.

Dôsledky diaľkového ovládania

```
String dunčo = "dunčo";
String dunčo2 = "dunč" + "o";

if(dunčo == dunčo2) {
     //ráno platí, večer už nie
}
```



- Podmienka platí, lebo *dunčo* aj *dunčo2* nemusia ukazovať na ten istý reťazec.
- Kompilátor nemusí vedieť, že *"dunč" + "o"* má nasmerovať na existujúci reťazec.
- Preto porovnávame cez *equals()* !

Zásada s veľkým Z

PAZ1C

Morálne ponaučenie:

- **objekty** porovnávame cez `equals()`!

názov typu objektu sa začína veľkým písmenom

  - *`String`y sú objekty!*

- **primitív**ne typy porovnávame cez `==`!

názov primitívu sa začína malým písmenom

  - primitívy nie je možné porovnávať cez equals()!
  - primitív nemá metódy, nastane kompilačná chyba

Zásada s veľkým Z_2

PAZ1C

Morálne ponaučenie 2:

- objekty porovnávame cez **==** jedine v prípade, že ho porovnávame s **null**.

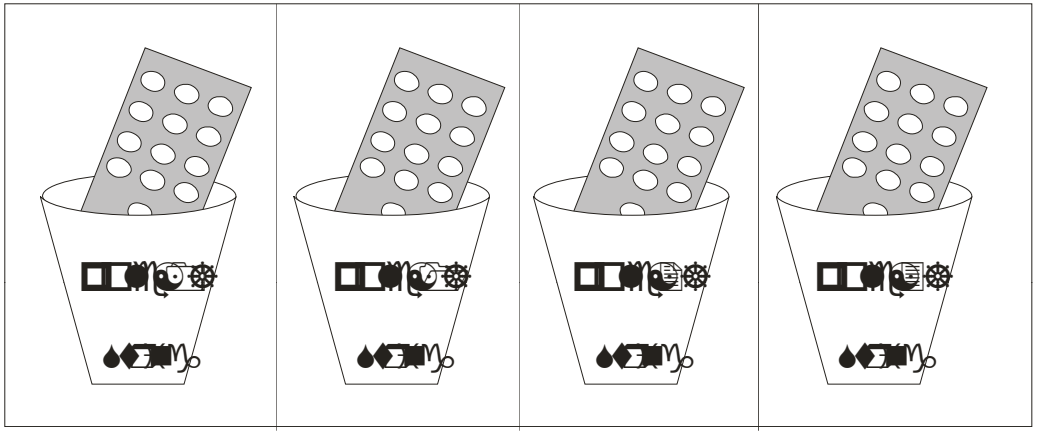
```
Pes pes; // v psovi je null

if(pes == null) {
  System.out.println("Kde je pes?“);
}
```

- nepíšeme ~~`pes.equals(null)`~~
  - vedie to k chybe
  - null znamená *nič* a *nič* nemá žiadne schopnosti (= žiadne metódy!)

Dôsledky diaľkového ovládania - polia

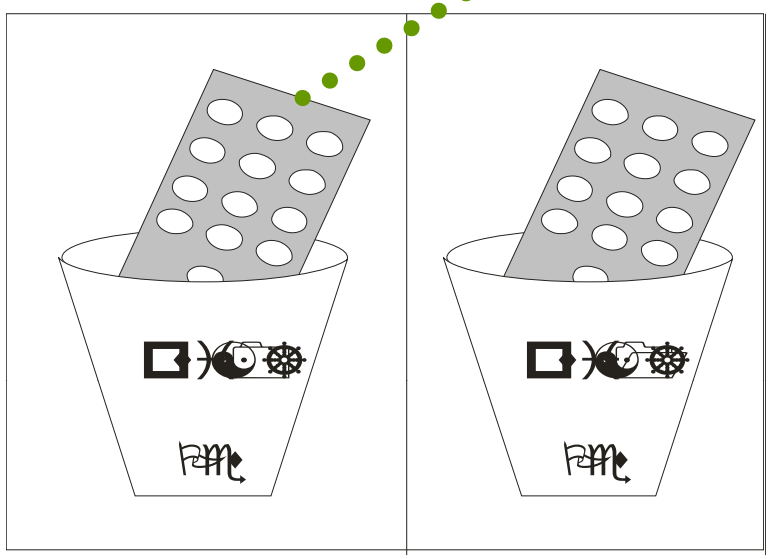

```
String[] pole = new String[4];
```



- Máme pole štyroch pohárikov s diaľkovými ovládaniami, ktoré neriadia žiaden objekt (neukazujú nikam)
- Každý z prvkov má hodnotu **null**.
- Dôsledok: `pole[0].length()` = `NullPointerException` = výbuch = snažíme sa volať metódu na neexistujúcom objekte

Dôsledky diaľkového ovládania – polia a objekty


```
Pes dunčo = new Pes();
Pes[] psi = new Pes[2];
psi [0] = dunčo;
psi[0].setVek(4);
System.out.println(psi[0].getVek());
System.out.println(dunčo.getVek());
```



Národnostné konflikty v triedach

- dosiaľ sme mali *jednoduché názvy* tried – Pes, Veľryba...
- problém nastáva, keď dva projekty pomenujú triedu rovnako
  - `Attribute`: spracovávač webových stránok v HTML
  - `Attribute`: v podpore pre tlač
  - `Attribute`: v projekte IGAP T. Horvátha
- Čo keď chcem používať vo svojej aplikácii aj HTML spracovávač aj IGAP?
  - *Riešenie 1*: dohoda medzi programátormi, premenovanie tried
    - HTMLAttribute, PrintAttribute, IgapAttribute
  - problém: s T. Horváthom sa dohodnete, s autormi Javy už nie

Zabaľme konflikty

Riešenie 2: balíčky

- zaveďme hierarchickú štruktúru á la adresáre pre triedy
- každá trieda má názov získaný z hierarchie
  - ***`org.htmlparser.Attribute`***: v HTML parseri
  - ***`javax.print.attribute.Attribute`***: v podpore pre tlač
  - ***`uinf.wid.Attribute`***: v projekte IGAP T. Horvátha

Zabaľme konflikty

- Ak chceme vytvoriť novú inštanciu HTML atribútu, musíme uviesť celý názov cesty.

```
org.htmlparser.Attribute atribút = new
   org.htmlparser.Attribute();
```

- Problém: dá si niekto na obed toto?

```
org.springframework.aop
.framework.autoproxy.metadata
.AttributesThreadLocalTargetSourceCreator c
   = new org.springframework.aop
         .framework.autoproxy.metadata
         .AttributesThreadLocalTargetSourceCreator();
```

Syndróm karpálneho tunelu a jeho riešenie

- namiesto klepkania názvu triedy použijeme *import*


```
import
public class AttributeTester {
   public static void main(String[] args) {
      uinf.wid.Attribute a = new uinf.wid.Attribute();
      Attribute a = new Attribute();
   }
}
```

- import hovorí toto: ak používam triedu `Attribute`, znamená to, že vlastne používam triedu `uinf.wid.Attribute`;
- *import* nie je *include*: obsah triedy `Attribute` sa **ne**vkladá do triedy `AttributeTester`

Balíčky

- každá trieda prináleží do nejakého balíčka
- výnimka-nevýnimka: existuje ***implicitný balíček*** (*default package*), ktorý je tvorený „koreňovým adresárom" hierarchie
- náš `Pes` bol zatiaľ v implicitnom balíčku
- triedy ***v implicitnom balíčku netreba importovať***
  - trieda v balíčku nevidí implicitný balíček
- v implicitnom balíčku nie sú štandardne žiadne triedy (pokiaľ nepoužívame *prasácky* napísané knižnice)

Štábna kultúra

- Odteraz zaradíme každú našu triedu do balíčka
- Príklad: `novotnyr.zvierata.Pes`
- Ale ako?

PAZ1C

```
package novotnyr.zvierata;

public class Pes {
     private String rasa;
     private int vek;
     ...
}
```

To však nie je všetko...

Balíčky

- Zabudli sme sa spýtať: ***čo napr. taký String?***
  - String sa volá `java.lang.String`
  - Triedy v balíčku `java.lang` sa nemusia importovať
- Trieda v rovnakom balíčku sa nemusí importovať

PAZ1C

```
package novotnyr.zvierata;

public class Pes {
```

```
package novotnyr.zvierata;

import novotnyr.zvierata.Pes;

public class PesTester {
        ...
        Pes pes = new Pes();
```

netreba

Balíčky

- Viacero importov z rovnakého balíčka vieme zapísať skrátene

```
import novotnyr.zvierata.Pes;
import novotnyr.zvierata.Mačka;
import novotnyr.zvierata.Lasica;

public class PesTester {
...
```

```
import novotnyr.zvierata.*;

public class PesTester {
...
```

importuj všetky triedy z balíčka `novotnyr.zvieratá`

Nadradené a podr(i)ad(e)né balíčky

- Trieda automaticky importuje (= netreba písať import) len triedy zo svojho balíčka
- Musíme importovať aj triedy z nadradených, aj z podriadených balíčkov

import triedy z podriadeného balíčka

import triedy z nadriadeného balíčka

```
package novotnyr;

import novotnyr.zvierata.Pes;

public class Búda {
  private Pes obyvateľ;
```

```
package novotnyr.zvierata;

import novotnyr.Búda;

public class Pes {
   private Búda bydlisko
```

Nadradené a podr(i)ad(e)né balíčky

- Hviezdičkový import znamená len *„importuj triedy z tohto balíčka"*
- Triedy z podriadených balíčkov sa takto **NEimportujú**
  - `import java.util.*` importne java.util.ArrayList, java.util.HashMap, atď
  - NEimportne napr. triedu `java.util.prefs.Preferences`, ani napr. `java.util.regex.Pattern`
- nedá sa ani použiť `import java.util.*.*.*`

Štábna kultúra

- Zdrojové súbory tried sa doteraz váľali v jednom adresári.
- Norma: skompilované triedy musia byť uložené v adresárovej štruktúre zodpovedajúcej hierarchii balíčkov
- Príklad: skompilovaná trieda `novotnyr.zvierata.Pes` musí byť v súbore

  ***nejakýAdresár*\novotnyr\zvierata\Pes.class**

Balíčky, CLASSPATH a iné potvory

- Ako Java vie, že triedu `novotnyr.zvierata.Pes` má načítať práve z tohto adresára?

  ***nejakýAdresár*\novotnyr\zvierata\Pes.class**

- premenná prostredia CLASSPATH

  WinXP: Ovládacie panely | Systém | Upresniť | Premenné prostredia | Systémové premenné

- obsahuje adresáre, v ktorých sa má začať prehľadávať hierarchia balíčkov

Balíčky, CLASSPATH a iné potvory

- Príklad:
  - majme `CLASSPATH=C:\`
  - chceme používať triedu `novotnyr.zvierata.Pes` uloženú v `D:\Java\psi\novotnyr\zvierata\Pes.class`
- Riešenie
  - prehľadávame CLASSPATH:
    - hľadáme súbor `C:\novotnyr\zvierata\Pes.class`
  - taký súbor neexistuje = **chyba!**

```
java.lang.NoClassDefFoundError: novotnyr.zvierata.Pes
```

Balíčky, CLASSPATH a iné potvory

- Riešenie
  - CLASSPATH upravíme na

    `CLASSPATH=C:\;D:\Java\psi`
  - prehľadávame CLASSPATH
    - `C:\novotnyr\zvierata\Pes.class` nenájdený
    - `D:\Java\psi\novotnyr\zvierata\Pes.class` nájdený = OK!
- Nenastavený CLASSPATH je to isté ako

  `CLASSPATH=.`

BODKA = Hľadaj v aktuálnom adresári

Balíčky, CLASSPATH a iné potvory

- CLASSPATH môžeme nastaviť aj pri spúšťaní java.exe

```
java -cp C:\;D:\Java\psi
   novotnyr.zvierata.Pes
```

- Integrované vývojové prostredia riešia CLASSPATH sami