Programovanie, algoritmy, zložitosť (UINF / PAZ1c)

Diel V.

Róbert Novotný
robert.novotny@upjs.sk
22. 10. 2008

Dedíme aj inštančné premenné

- každá trieda zdedí okrem metód aj inštančné premenné.

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

```
class CirkusovýPes extends Pes {
   public String toString() {
        return "Cirkusant " + rasa;
   }
}
```

The variable `Pes.vek` is not visible!
/
Premenná `Pes.vek` nie je viditeľná!

Táto premenná by sa mala zdediť.

paz1c

Dedíme aj inštančné premenné

- spomeňme si na privátne premenné
- ***k privátnym premenným môže pristupovať len kód v danej triede***
- riešenie: **chránené (protected)** premenné
- k *protected* premenným môže pristupovať
  - kód v danej triede
  - kód v zdedených triedach
  - kód v triedach, ktoré sa nachádzajú v rovnakom balíčku

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

```
class CirkusovýPes extends Pes {
   public String toString() {
        return "Cirkusant " + rasa;
   }
}
```

paz1c

Viditeľnosť

- verejné (public) premenné sa tiež zdedia
  - také premenné by však v triede nemali byť.
- kedy *private* a kedy *protected*?
  - privátne inštančné premenné sú určené len a len danej triede.
  - nepredpokladá sa, že k nim budú mať prístup zdedené triedy
  - chránené sú určené ,,blízkej rodine" a dedičom
- rady starej matere: je lepšie označiť premennú ako *private* a v prípade potreby ju zmeniť na *protected*

Viditeľnosť premenných

Príklad: trieda Otec a inštančná premenná Pivo

- privátna (*private*) = len otec môže piť pivo
- chránená (*protected*) = pivo môže vypiť len otec a môže mu ho vypiť synček alebo iný člen rodiny (= balíčka)
- verejná (*public*) = pivo môže vypiť otcovi hocikto

Príklad použitia v projekte *JukeBox*

```
public abstract class JukeBox {
  protected String meno;
  public void hrajVšetky() {
     for(CD cd : getCDs()) {
         cd.zahraj();
     }
  }
  protected abstract ArrayList<CD> getCDs();
}
```

- Všetcí prípadní potomkovia JukeBox-u budú mať meno
- JukeBox vie prehrať CDčka, ktoré v ňom sú
- To, odkiaľ sa získa zoznam CDčiek, sa prenechá na potomkov

Príklad použitia v projekte *JukeBox*

```
public class SúborovýJukeBox extends JukeBox {
  protected ArrayList<CD> getCDs() {
     ArrayList<CD> cds = new ArrayList<CD>();
     File súbor = new File("C:/MP3");
     for(File podsúbor : súbor.list()) {
         CD cd = new CD();
         cd.setNázov(podsúbor.getName());
         cds.add(cd);
     }
     return cds;
  }
}
```

vypíše súbory v danom adresári

názov súboru

Príklad použitia v projekte *JukeBox*

```
public class DatabázovýJukeBox extends JukeBox {
  protected ArrayList<CD> getCDs() {
     Databáza db = Databáza.getDatabáza();
     ArrayList<CD> cds = db.vykonajDopyt("SELECT *
         FROM cd ORDER BY 1");
     return cds
  }
}
```

nejaká zázračná trieda pracujúca s SQL databázou

Príklad použitia v projekte *JukeBox*

```
public static void main(String[] args) {
  JukeBox j = new JukeBox();
  j.hrajVšetko();
  j = new SúborovýJukeBox();
  j.hrajVšetko()
  j = new DatabázovýJukeBox();
  j.hrajVšetko();
}
```

Cannot instantiate type / Nemôžem vytvoriť inštanciu typu

Dedenie konštruktorov

- Trieda zdedí od svojho predka stav a chovanie (t. j. metódy a inštančné premenné)
  - za predpokladu, že nie sú *private*
  - cirkusový pes zdedí schopnosti štekať od klasického psa
- Čo s konštruktormi?
  - **konštruktory sa nededia!**

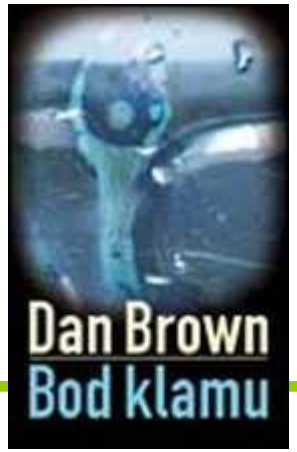
Dedenie konštruktorov

```
public class Bod {
    protected double x;
    protected double y;

    //..gettery a settery
}
```

```
Bod b = new Bod();
b.setX(25.0);
b.setY(-3.0);
```

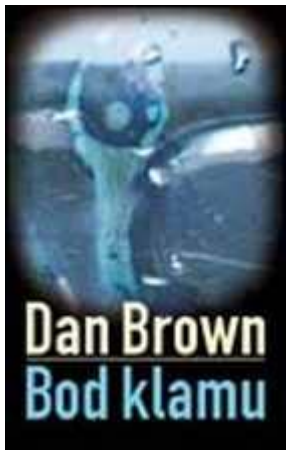
- Používateľovi uľahčíme používanie triedy Bod, ak zavedieme konštruktor s dvoma parametrami



Dedenie konštruktorov

```
public class Bod {
    protected double x;
    protected double y;

    public Bod(double x, double y) {
      this.x = x;
      this.y = y;
    }
    //..gettery a settery
}
```




```
Bod b = new Bod(25.0, -3.0);
```

ušetrili sme dva riadky, a je to prehľadnejšie

Dedenie konštruktorov

```
Bod b = new Bod(25.0, -3.0);
```

Čo keď chceme používať predošlý spôsob?

```
Bod b = new Bod();
b.setX(25.0);
b.setY(-3.0);
```

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

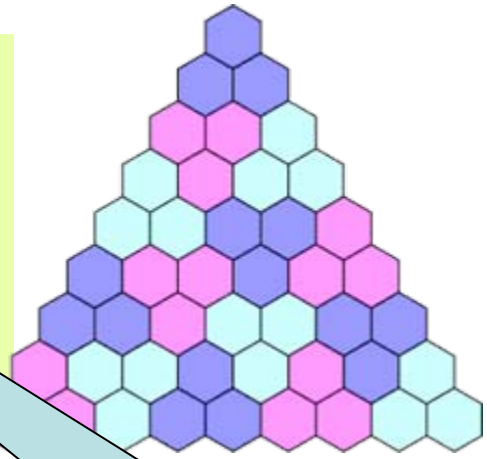
```
public class Bod {
    ...
    public Bod() {
      //prazdny konstruktor
    }
    public Bod(double x, double y) {
       this.x = x;
       this.y = y;
    }
    ...
}
```

Dedenie konštruktorov

Chceme zaviesť farebné body s využitím triedy `Bod`

```
public class FarebnýBod extends Bod {
   private String farba;

   //gettery a settery
}
```



```
FarebnýBod fb = new FarebnýBod();
b.setX(25.0);
b.setY(-3.0);
b.setFarba("žltá");
```

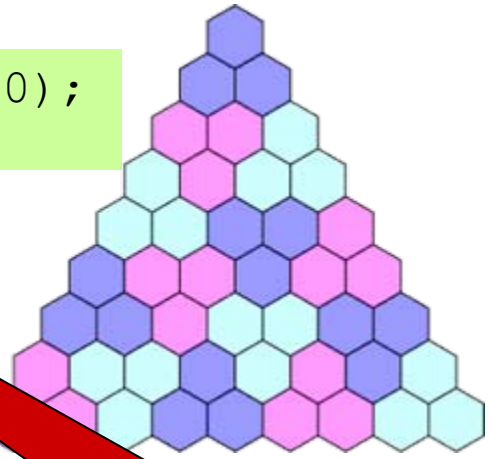
trieda nezdedila
žiadne konštruktory

má len jeden
prázdny konštruktor

Dedenie konštruktorov

Chceme zaviesť farebné body s využitím triedy `Bod`

```
FarebnýBod fb = new FarebnýBod(25.0, -3.0);
```



Dodáme dva konštruktory, tak, ako to bolo v `Bod`-e

```
public FarebnýBod() {
    //prazdny konstruktor
}
public FarebnýBod(double x, double y) {
   this.x = x;
   this.y = y;
}
```

trieda nezdedila žiadne konštruktory

má len jeden **prázdny** konštruktor

zdedená premenná

Dedenie konštruktorov

Všimnime si: v druhom konštruktore `FarebnéhoBodu` robíme presne to, čo v druhom konštruktore `Bodu` – zbytočne sa opakujúci kód

```
public FarebnýBod(double x, double y) {
   this.x = x;
   this.y = y;
}
```

Vieme však zavolať rodičovský konštruktor – **`super()`**

```
public FarebnýBod(double x, double y) {
   super(x, y);
}
```

volaj rodičovský dvojparametrový konštruktor

Dedenie konštruktorov

Pravidlo: ak používame rodičovského konštruktora cez `super()` musíme to spraviť ako prvú vec v metóde

„To je super!"

```
public FarebnýBod(double x, double y) {
   super(x, y);
   System.out.println("Nový bod hotový");
}
```

Chyba – nesprávne použitie:

```
public FarebnýBod(double x, double y) {
   System.out.println("Nový bod hotový");
   super(x, y);
}
```

Dedenie konštruktorov

- **Pravidlo:** ak používame rodičovského konštruktora cez `super()` musíme to spraviť ako prvú vec v metóde.
- Prečo?
  - Dôvodom je inicializácia premenných.
  - Rodičovská trieda musí byť pripravená na použitie ešte pred inicializáciou potomkov.
  - ***„Na to, aby ste sa mohli narodiť vy, potrebujete mať živých a zdravých rodičov"***

Dedenie konštruktorov

Ďalší príklad: dopracujme ešte jeden konštruktor

```
public FarebnýBod(double x, double y, String farba) {
   super(x, y);
   this.farba = farba;
}
```

- Premenné pre súradnice sme nainicializovali pomocou konštruktora `Bod`-u.
  - Obyčajný bod dokáže nastaviť svoje súradnice už pri vytvorení, tak prečo to nevyužiť?
- V potomkovi už len donastavujeme farbu

Dedenie konštruktorov

Otázka: Vieme, že `Bod` dedí z `Object`-u a napísali sme mu konštruktor. Prečo sme v ňom nepísali **`super()`**?

```
public Bod(double x, double y) {
   //prazdny konstruktor
}
```

- Ak v konštruktore nepoužijeme **`super()`**, Java automaticky dodá volanie rodičovho prázdneho konštruktora

```
public Bod(double x, double y) {
   super();
}
```

Automaticky dodané

Dedenie konštruktorov

Chyba! Implicitný rodičovský konštruktor `Človek()` nie je definovaný! Prečo?

```
public class Človek{
   public Človek(String meno) {
        System.out.println(meno);
   }
}
```

V `Človek`-u je definovaný len jeden konštruktor. Prázdny konštruktor v ňom nie je.

```
public class Študent extends Človek {
   public Študent(String meno) {
        super();
        System.out.println(meno);
   }
}
```

V `Študent`-ovi je tiež definovaný len jeden konštruktor. Prázdny konštruktor v ňom tiež nie je.

Java si ,,domyslí" volanie rodičovského konštruktora bez parametrov

Rodičovský konštruktor (teda konštruktor bez parametrov v triede `Človek`) však neexistuje!

Dedenie konštruktorov

Ale pozor:

```
public class Človek {
   public Človek(String meno) {
        System.out.println(meno);
   }
}
```

Chyba! Implicitný rodičovský konštruktor `Človek()` nie je definovaný!

- Java automaticky dodá volanie rodičovského konštruktora bez parametrov - super().
- V `Človek`-u však **neexistuje** bezparametrický konštruktor
- Neexistuje teda možnosť, ako inicializovať a pripraviť `Človeka` skôr, než `Študenta`

```
public class Študent extends Človek {
   public Študent(String meno) {
        super(meno);
        System.out.println(meno);
   }
}
```

výpis už netreba, vykoná sa v rodičovi

Musíme sem výslovne uviesť `super(meno)`

paz1c

Dedenie konštruktorov - zhrnutie

- Konštruktory sa na rozdiel od metód nededia
- Pri volaní konštruktora sa zavolá rodičovský konštruktor, ktorý zavolá rodičovský konštruktor, ..., až kým neskončíme v `Object`-e
- Volanie sa deje cez `super()` – buď jeho explicitným uvedením alebo automaticky

Spravia všetko a potom zavolajú rodičov

...a potom rodičov ich rodičov

...a potom rodičov ich rodičov ICH rodičov

Pretypovanie

- Ak máme hierarchiu tried, tak môžeme priraďovať len potomka do predka

```
ZobcováFlauta z = new ZobcováFlauta();
Flauta f = new Flauta();
```

f = z

~~z = f~~

- existujú však prípady, keď môžeme priradiť predka do potomka
- príklad: staré zoznamy v JDK 1.4
  - metóda get() na ArrayList-e vracala Object

```
ArrayList zoznam = new ArrayList();
zoznam.add(new Pes("Dunčo"))

Object psovitýObjekt = zoznam.get(0);
```

vkladám reťazec

späť dostanem Object

Vráť mi môjho psíííška!

- Ako spravím z objektu psa?

  „*Ako zopsujem objekt?*"
- Použijeme pretypovanie
  - pretypovanie je zmena dátového typu
- Idea: ja som si na 100% istý, že som do zoznamu dával objekt typu `Pes` a teda môžem zmeniť `Object` na `Psa`

```
Object psovitýObjekt = zoznam.get(0);

Pes lietajúciPes = (Pes) psovitýObjekt;
```

menením dátový typ premennej

Pretypovanie

- Skrátený zápis:

```
Pes pes = (Pes) zoznam.get(0);
```

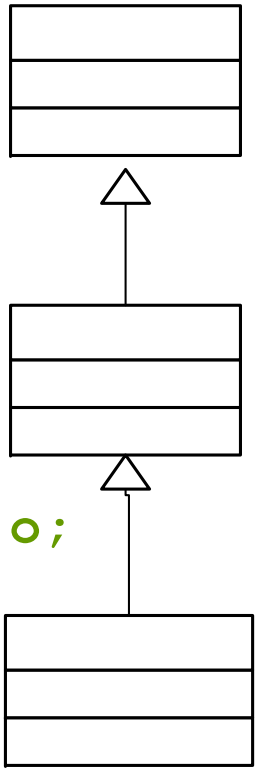
- Pretypovávať je správne len vtedy, ak objekt z ktorého pretypovávame je naozaj typu, na ktorý pretypovávame
  - Pretypovanie je správne len vtedy, ak prvok vytiahnutý zo zoznamu je naozaj typu `Pes`.
- Strach a hrôza: ak pretypujeme nesprávne, dozvieme sa to až pri behu programu
  - `java.lang.ClassCastException` – výnimka
  - nesprávne pretypovanie v C/C++: nádherné obskúrne chyby

Pretypovanie

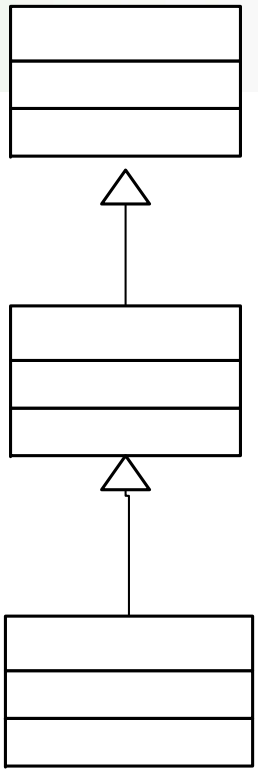
- Kedy pretypovávame?
  - ak máme objekt typu `Rodič` a chceme zmeniť jeho typ `Potomok`, pričom `Potomok extends Rodič`

```
CirkusovýPes cp = new CirkusovýPes();
Pes obyčajnýPes = cp;
CirkusovýPes c2 = (CirkusovýPes) obyčajnýPes;

Object o = c2;
Pes ďalšíObyčajnýPes = (Pes) o;
CirkusovýPes ešteJedenCirkusant = (CirkusovýPes) o;
```



Pretypovanie smerom nahor



- Pretypovanie smerom nahor sa deje automagicky

```
CirkusovýPes cp = new CirkusovýPes();
Pes obyčajnýPes = (Pes) cp;
```

- Pretypovanie netreba písať

```
CirkusovýPes cp = new CirkusovýPes();
Pes obyčajnýPes = cp;
```

- To sme videli aj v prípade hudobných nástrojov.
- Pretypovávať môžeme aj ,,kompatibilné" primitívy:

```
float pi = 3.14f;
int celéPi = (int) pi;
```

odreže sa desatinná časť

```
double pi = 3.14;
float pi2 = (float) pi;
```

zníži sa presnosť

Zisťovanie typu premennej

- Niekedy je vhodné zistiť, či bude pretypovanie bezpečné
- Operátor **instanceof** zistí, či má premenná daný typ alebo niektorý z rodičovských typov

```
Pes obyčajnýPes = new Pes();
if(obyčajnýPes instanceof CirkusovýPes) {
  CirkusovýPes c2 = (CirkusovýPes) obyčajnýPes;
}
```

obyčajnýPes je ***Psom***, ale nie je ***CirkusovýmPsom***

```
CirkusovýPes pes = new Pes();
if(pes instanceof Object) {
  ...
}
```

pes ***je*** Objectom

Výnimky znovu útočia

```
BufferedReader br = null;
try {
   br = new BufferedReader(new FileReader("C:/test.txt");
   ...
} catch (FileNotFoundException e) {
   System.out.println("Súbor nebol nájdený");
} catch (IOException e) {
   System.out.println("Chyba pri čítaní!");
} finally {
   if(br != null) {
     try {
         br.close();
     } catch (IOException e) {
     }
   }
}
```

chutné, však?

Ako vyhodiť vlastnú výnimku

- výnimky sú objektami
- aj my môžeme vyhadzovať výnimky

```
void setVek(int vek) throws ZápornýVekException
{
   if(vek < 0) {
      ZápornýVekException e = new ZápornýVekException();
      throw e;
   }
}
```

vyhlasujem, že v tejto metóde môže nastať chyba!

vyhoď výnimku!

- pravidlo: každá výnimka, ktorá môže nastať, musí byť uvedená v hlavičke metódy

Ako vyhodiť vlastnú výnimku

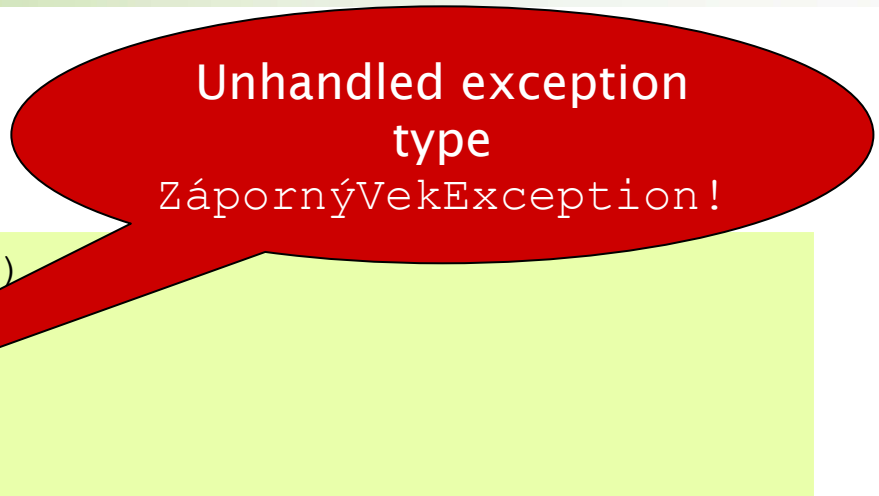
- výnimky sú objektami
- aj my môžeme vyhadzovať výnimky

```
public class ZápornýVekException extends Exception {
   //tu nič nie je
}
```

- vytvorili sme vlastnú výnimku
- výnimka môže mať
  - vlastné inštančné premenné
  - vlastné metódy
  - konštruktory...

Ako odchytiť vlastnú výnimku

- príklad ošetrenia vlastnej výnimky




```
public static void main(String[] args)
  Pes pes = new Pes();
  pes.setVek(-2000);
}
```

```
public static void main(String[] args) {
  try {
    Pes pes = new Pes();
    pes.setVek(-2000);
  } catch (ZápornýVekException e) {
    System.out.println("Psovi nemožno nastaviť záporný vek!");
  }
}
```

Chyť alebo ohlás ďalej

- ak naša metóda výnimku neošetruje, môže ju posunúť ďalej volajúcej metóde – výnimka vybuble vyššie
- platí pravidlo:
  - výnimku musíme **buď odchytiť** v `catch` bloku
  - alebo ju môžeme neošetriť a **poslať ďalej**

„systém padajúceho..."

- výnimku pošleme ďalej tak, že ju uvedieme v `throws` klazule v hlavičke metódy

ak nastane problém, niekto ho vyriešiť musí!

Pohadzujeme výnimky hore-dole

- ak naša metóda výnimku neošetruje, môže ju posunúť ďalej volajúcej metóde – výnimka vybuble vyššie

```
public class Čitateľ {
  void načítaj() throws FileNotFoundException
  {
    String cesta = "C:/test.txt";
    FileReader r = new FileReader(cesta);
  }
}
```

táto metóda hádže FileNotFoundException

ošetrenie necháme na niekoho iného

Neošetrená výnimka FileNotFoundException