

# 7

## Dedičnosť II

# Pojmy zavedené v 6. prednáške<sub>(1)</sub>

- odstránenie duplicít medzi triedami
  - skladanie – kompozícia
  - dedičnosť
- dedičnosť
  - typológia
- trieda dedí
  - vonkajší pohľad
  - vnútorný pohľad
  - nededí konštruktory

# Pojmy zavedené v 6. prednáške<sub>(2)</sub>

- interné ukrývanie informácií
- vzťahy pri dedičnosti
  - predok
  - potomok
  - priamy predok
  - priamy potomok
  - absolútny predok

# Pojmy zavedené v 6. prednáške<sub>(3)</sub>

- typy dedičnosti
  - jednoduchá – strom dedičnosti
  - jednoduchá – les dedičnosti
  - viacnásobná – mriežky dedičnosti

# Pojmy zavedené v 6. prednáške<sub>(4)</sub>

- porovnanie skladania a dedičnosti
  - znovupoužitelnosť
  - implementačná závislosť
- dedičnosť a interface

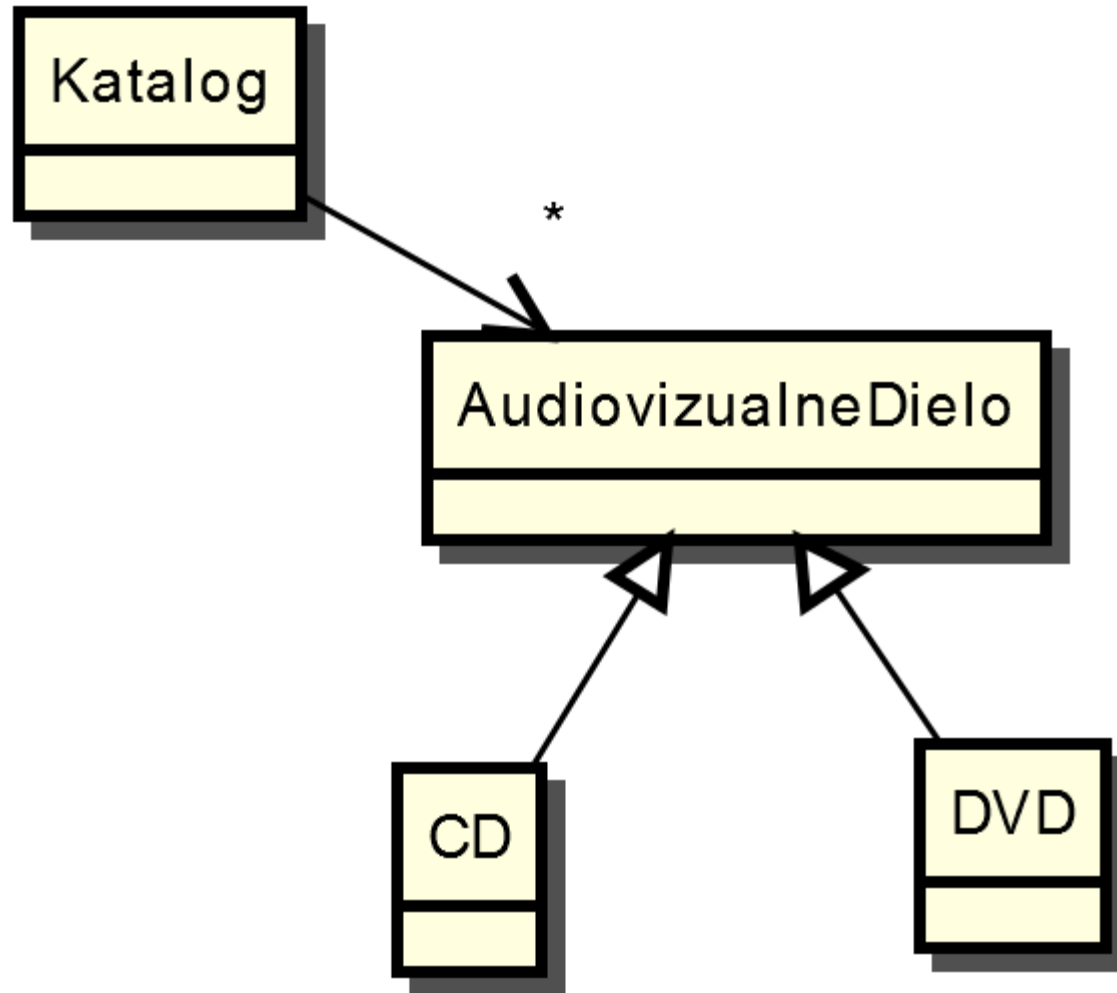
# Pojmy zavedené v 6. prednáške<sub>(5)</sub>

- dedičnosť a typová kompatibilita
- dedičnosť a polymorfizmus
- prekrývanie metód
  - kľúčové slovo super
- jazyky – implementácia polymorfizmu

# Cieľ prednášky

- abstraktná trieda
- vzťahy is-a, has-a
- extenzia triedy
- Liskovej princíp substitúcie
- trieda Object
  
- príklad: KCalB

# KCaIB – súčasný stav






# Abstraktná trieda – motivácia

- inštancie vytvorené triedou AudiovizualneDielo
  - nemajú význam
  - v skutočnosti neexistujú
  - vytvárať je technicky možné
  - [abstraktná trieda](#)
- trieda AudiovizualneDielo
  - prvotne odstránenie duplicity v triedach
  - predok pre rôzne typy avi diel
  - koreň hierarchie typov (domény)

# Abstraktná trieda – vlastnosti

- charakteristická vlastnosť – nevytvára inštancie
- podľa jazyka
  - dobrovoľná – možno vytvoriť, nemá význam 
  - povinná – nemožno vytvoriť, zákaz
- Java – možnosť označiť triedu ako abstraktnú

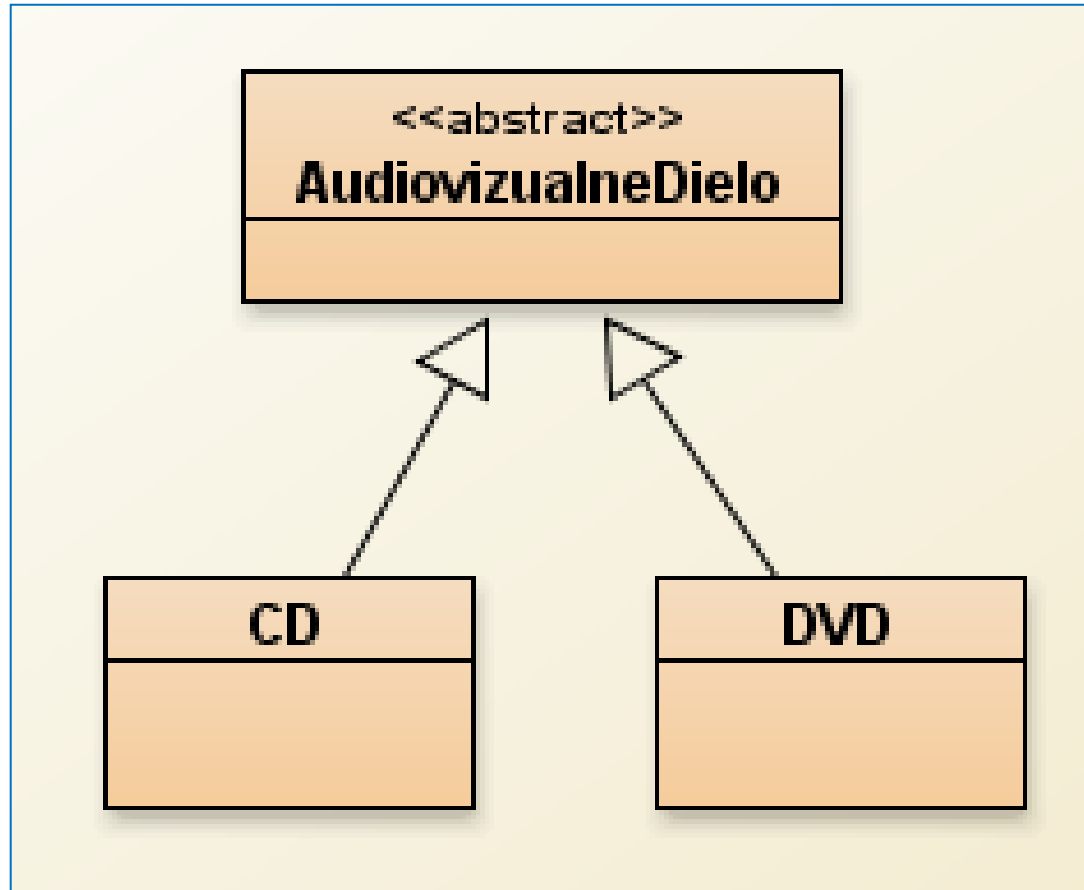
# Konkrétna trieda

- inštancie vytvorené triedami CD a DVD
  - majú význam
  - existujú aj v skutočnosti
- konkrétna trieda
  - bežne vytvára inštancie

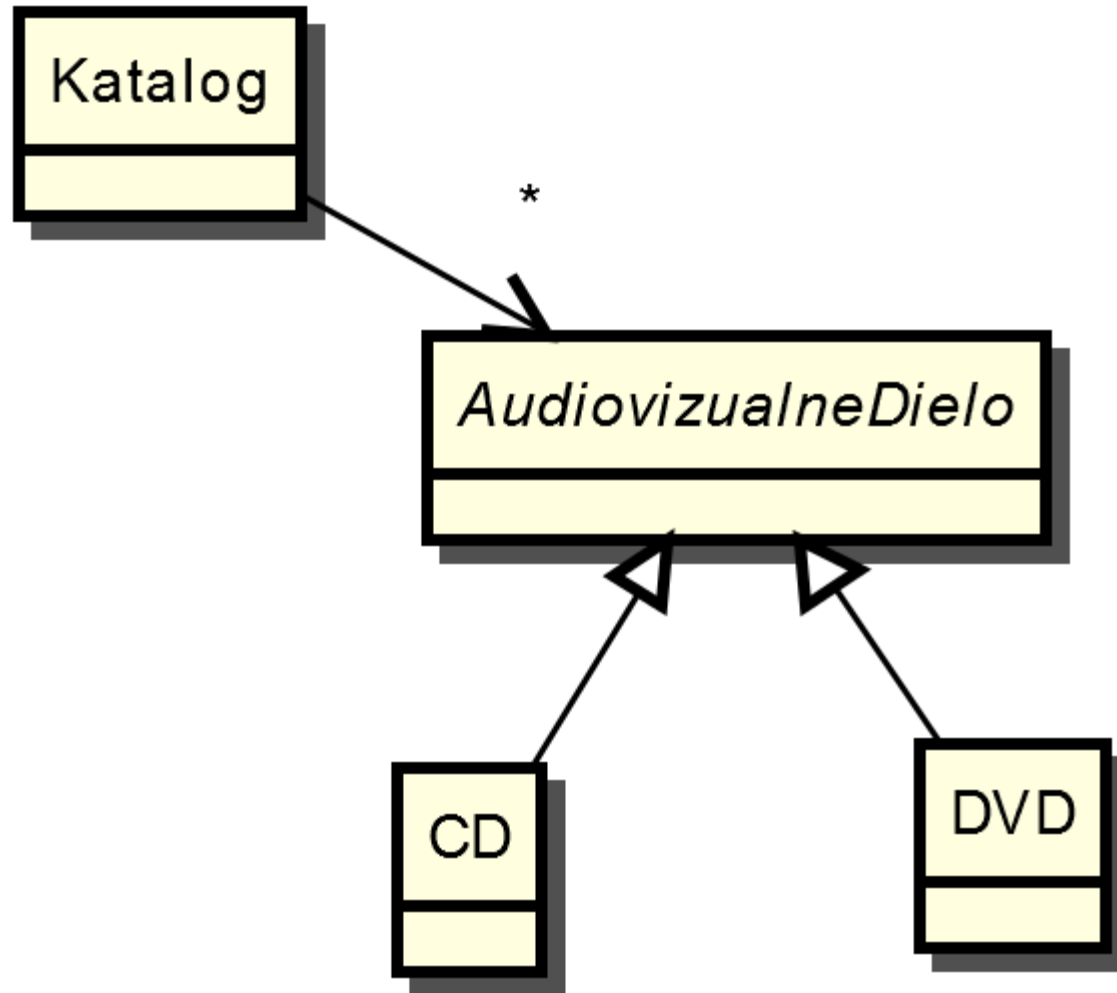
# Abstraktná trieda – modelovanie

- BlueJ
  - stereotyp <<abstract>>
  - nad menom triedy
- UML
  - názov triedy – kurzíva

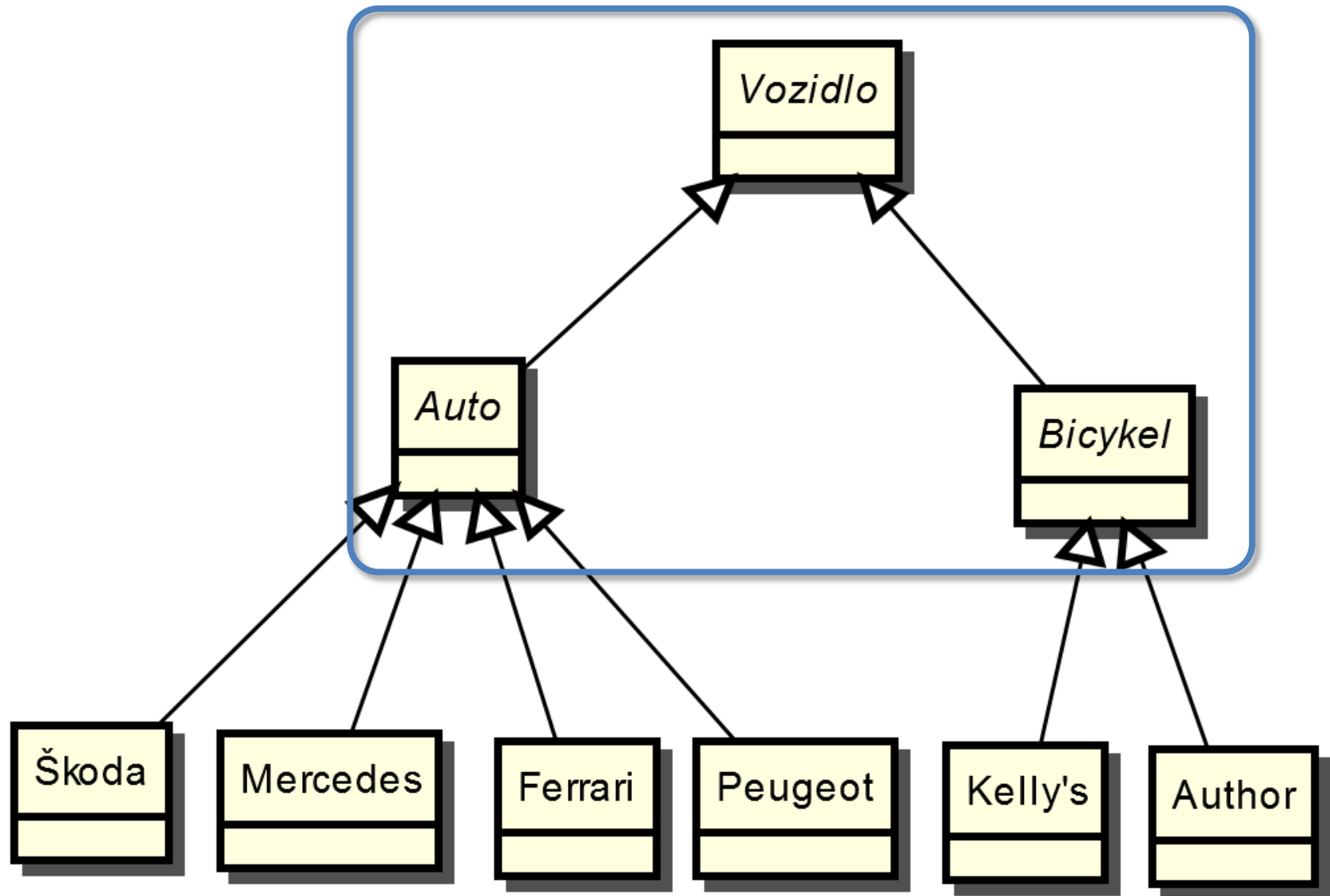
# Abstraktná trieda – BlueJ



# Abstraktná trieda – UML



# Abstraktné triedy v reálnom svete<sup>(1)</sup>



# Abstraktné triedy v reálnom svete<sub>(2)</sub>

- Trieda Vozidlo
  - definované chovanie (pohyb dopredu, odviešť človeka...)
  - nevieme si predstaviť jej inštanciu – vybavíme si konkrétne auto, alebo bicykel
  - abstraktná trieda



# Abstraktná trieda – Java

- kľúčové slovo **abstract**
- hlavička triedy

```
public abstract class AudiovizualneDielo  
{  
    // telo triedy  
}
```

# KCaIB – metódy vypis<sub>(1)</sub>

- AVIDielo – definícia metódy
- CD, DVD – prekrytie metódy
- polymorfizmus
- konečná podoba metódy – potomok

# KCaIB – metódy vypis

- AVIDielo – metóda nemá konečnú podobu
- CD, DVD – doplnenie informácie
  - môže vyžadovať iné usporiadanie informácií
- iné možnosti návrhu metódy v AVIDielo
  - prázdne telo metódy
  - abstraktná metóda

# Abstraktná a konkrétna metóda

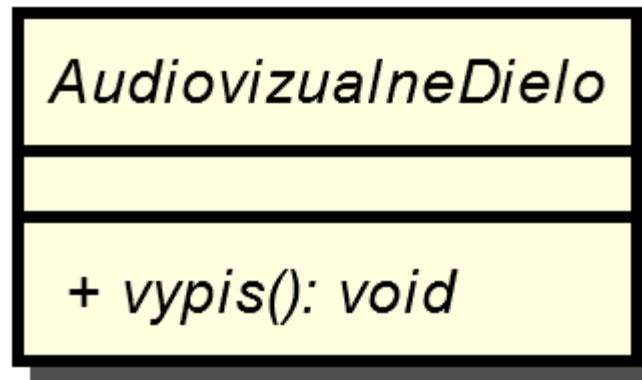
- metódy v abstraktnej triede
  - konkrétne
    - definícia tela metódy v tele triedy
    - definícia správy v rozhraní triedy
  - abstraktné
    - definícia správy v rozhraní triedy
- metódy v konkrétnej triede
  - len konkrétne
    - definícia tela metódy v tele triedy
    - definícia správy v rozhraní triedy

# Abstraktná a konkrétna metóda – príklad

- konkrétna metóda
  - bicykel ide dopredu po potiahnutí pedálmi – bez ohľadu na konkrétny typ
- abstraktná metóda
  - bicykel vie zabrzdiť – konkrétna implementácia (čelúšťové brzdy, V-brzdy, kotúčové brzdy, hydraulické brzdy...) závisí na konkrétnom type

# Abstraktná metóda – UML

- definícia metódy kurzívou



# Abstraktná metóda – Java

- jazyková konštrukcia
- zabezpečuje správu do rozhrania
- nemá žiadne telo
  
- kľúčové slovo **abstract** v hlavičke metódy
- hlavička ukončená bodkočiarkou

```
public abstract void vypis();
```

# AudiovizualneDielo – metóda vypis

```
public abstract class AudiovizualneDielo  
{  
    ...  
    public abstract void vypis();  
    ...  
}
```



# Implementácia abstraktnej metódy

- ako bežná metóda
- nemá zmysel prekrývanie abstraktnej metódy
  - neexistujúca metóda sa nedá prekryť

=>

- nepoužíva kľúčové slovo super
  - `super.abstraktnaMetoda()` – syntaktická chyba

# Abstraktná trieda vs. metóda

- abstraktná metóda – trieda musí byť abstraktná
- abstraktná trieda
  - nechceme vytvárať inštancie – voliteľná
  - definuje abstraktnú metódu – povinná
  - dedí abstraktnú metódu – povinná
  - implementuje abstraktnú metódu – podľa potreby

# CD – metóda vypis<sub>(1)</sub>

```
public class CD extends AVIDielo
{
    ...
    public void vypis()
    {
        // príkazy tela metódy
    }
    ...
}
```

# CD – metóda vypis<sub>(2)</sub>

```
System.out.println("CD:");  
System.out.println("    Autor: " + aAutor);  
System.out.println("    Titul: " + this.dajTitul());  
System.out.println();  
System.out.println("    Pocet skladieb: " +  
                    aPocetSkladieb +  
                    " (celkovo " +  
                    this.dajCelkovyCas() + " minut)");  
this.vypisKomentar();
```

# Vzťah is-a

- is-a – „is a“ – anglicky „je“
- dedičnosť je realizáciou vzťahu is-a
- každé CD „je“ audiovizuálne dielo
- =>
- každá inštancia CD je inštanciou  
AudiovizualneDielo

# Dedičnosť a extenzia triedy

- extenzia triedy
  - množina všetkých inšancií
  - vzťah „is-a“  $\Rightarrow$  aj inšancií potomkov
- dôsledok
  - abstraktné triedy môžu mať neprázdnu extenziu

# Operátor instanceof – extenzia

prvyOperand instanceof DruhyOperand

- vracia true
  - prvýOperand je inštanciou triedy DruhyOperand
- rozšírenie:
- vracia true
  - prvýOperand patrí do extenzie triedy DruhyOperand

# Vzťah has-a

- vyjadrenie vzťahu celku a jeho častí
- celok má časti
- obdoba vzťahu is-a pre kompozíciu (skladanie)
- DigitalneHodiny „má“ CiselnýDisplej



# CD – metóda vypis – komentáre

```
System.out.println("CD:");  
System.out.println("    Autor: " + aAutor);  
System.out.println("    Titul: " + this.dajTitul());  
System.out.println();  
System.out.println("    Pocet skladieb: " +  
                    aPocetSkladieb +  
                    " (celkovo " +  
                    this.dajCelkovyCas() + " minut)");  
this.vypisKomentar();
```

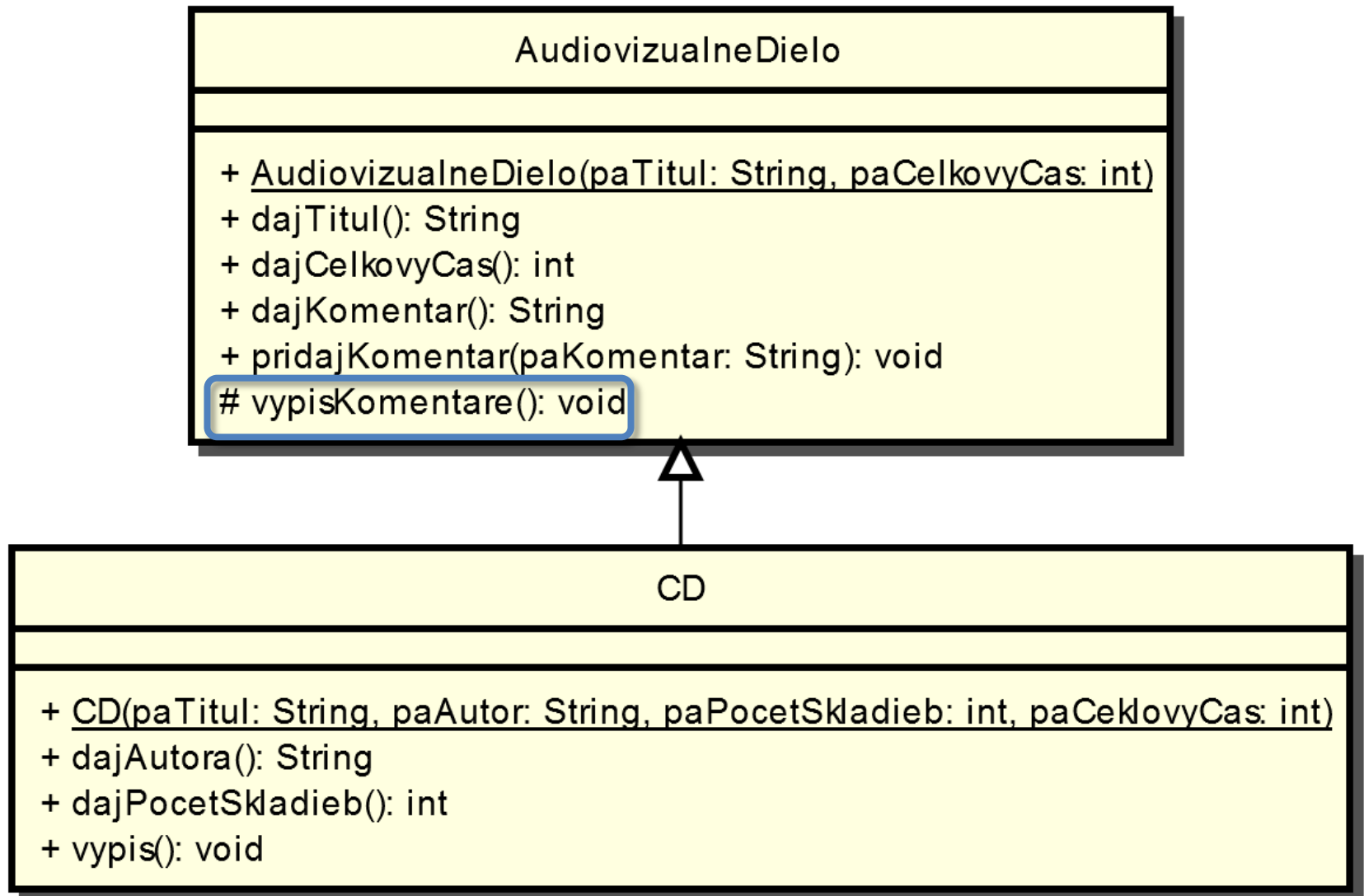
# Prístupové právo protected

- selektívne prístupové práva
  - „protekcia“ pre potomkov 😊
- **public** – verejný prístup všetkým iným objektom
- **private** – súkromný prístup samotného objektu
- **protected**
  - verejný pre každý objekt z extenzie
  - súkromný pre každý objekt mimo extenzie

# Protected – Java

```
protected void vypisKomentar()  
{  
    if (aKomentar != null)  
    {  
        System.out.println(aKomentar);  
    }  
}
```

# Protected – UML

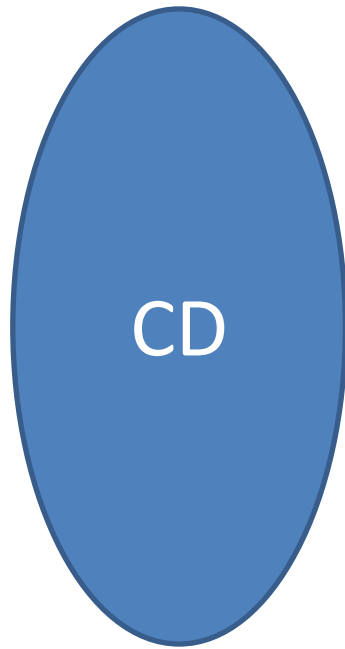


# Dedičnosť a návrh tried

- generalizácia
- špecializácia
- dedičnosť – vzťah „gen-spec“

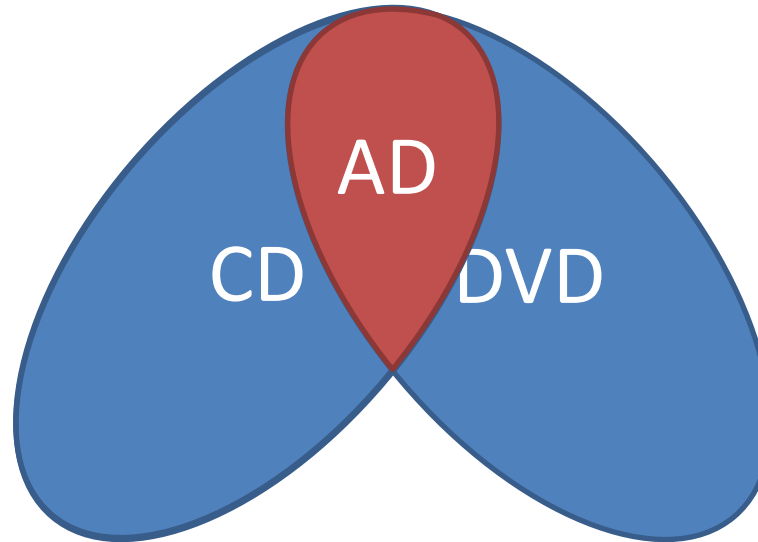
- spoločný predok z konkrétnych tried
- KCalB
  - trieda CD
  - trieda DVD
  - z nich trieda AudiovizualneDielo

# Samostatné triedy CD a DVD



# Vytvorenie spoločnej časti AVIDielo

generalizácia: (CD, DVD) → AudiovizualneDielo

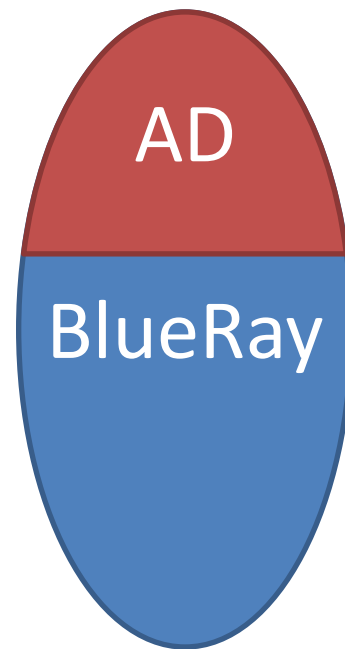




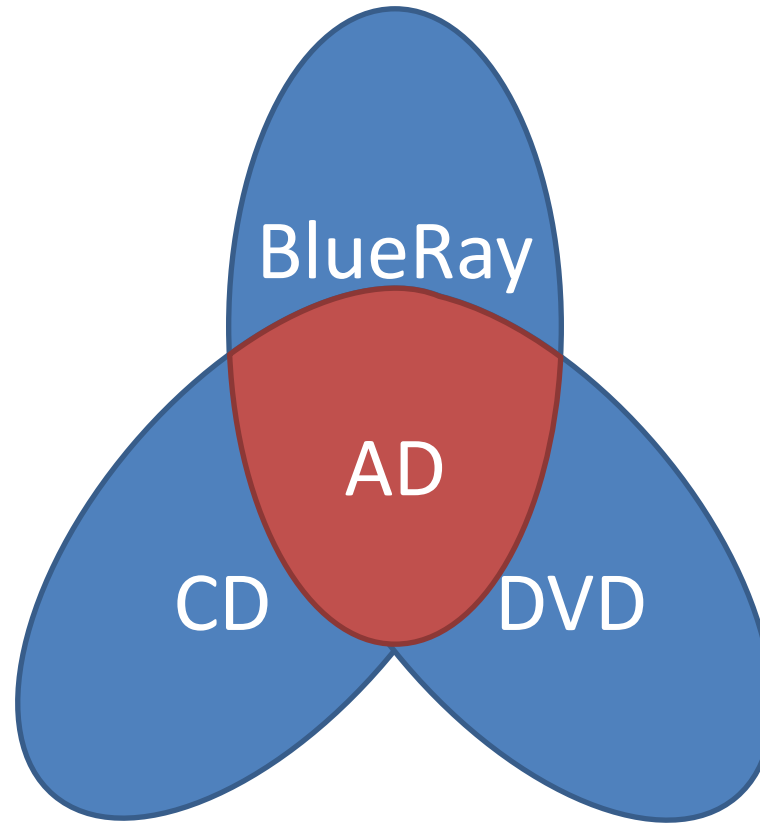
- odvodenie potomka z predka
- KCalB
  - trieda AudiovizualneDielo
  - z nej trieda BlueRay

# Nová trieda BlueRay<sub>(1)</sub>

špecializácia: AudiovizualneDielo → BlueRay



# Nová trieda BlueRay<sub>(2)</sub>

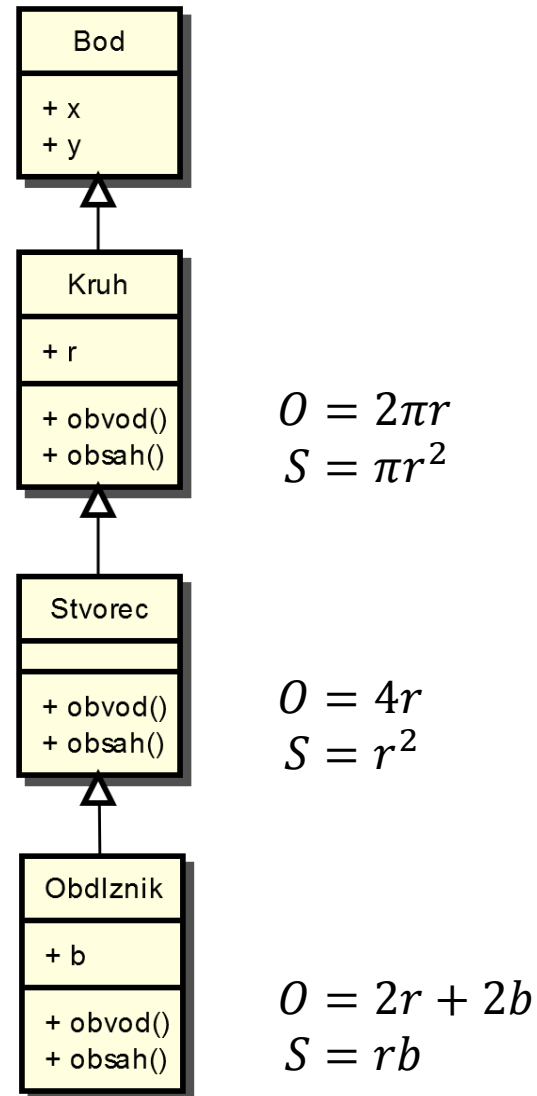


# Nebezpečenstvá dedičnosti

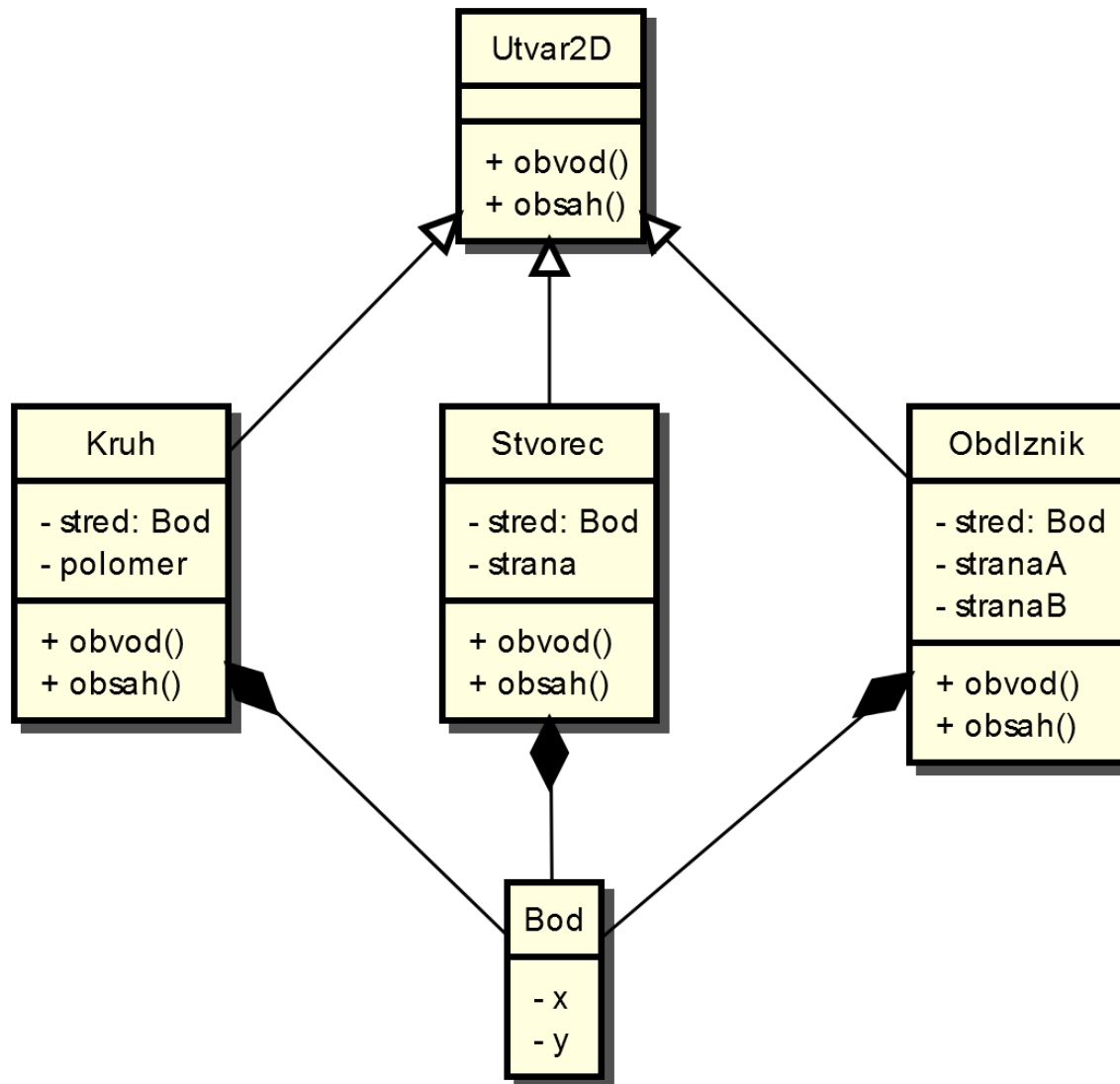
- explózia tried
- nelogické použitie (zneužitie) dedičnosti
- porušenie Liskovej princípu substitúcie
- zvyšovanie implementačnej závislosti



# „Technická“ dedičnosť



# „Logická“ dedičnosť



# Barbara Liskov(\*1939)



- 1987 – princíp substitúcie pre typy
- 2004 – medaila Jon von Neumana
- 2008 – Turingova cena

# The Liskov Substitution Principle (LSP)

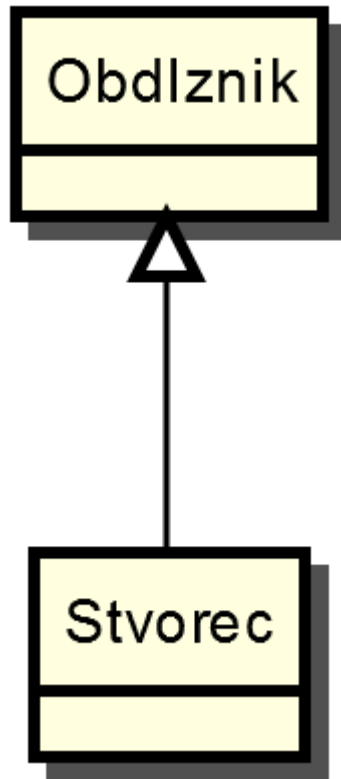
- *Let  $q(x)$  be a property provable about objects  $x$  of type  $T$ . Then  $q(y)$  should be true for objects  $y$  of type  $S$  where  $S$  is a subtype of  $T$ .*
- Nech  $q(x)$  je preukázateľná vlastnosť objektu  $x$  typu  $T$ . Potom  $q(y)$  by mala platiť pre objekt  $y$  typu  $S$ , kde  $S$  je podtyp typu  $T$ .



# Liskovej princíp substitúcie – slovensky

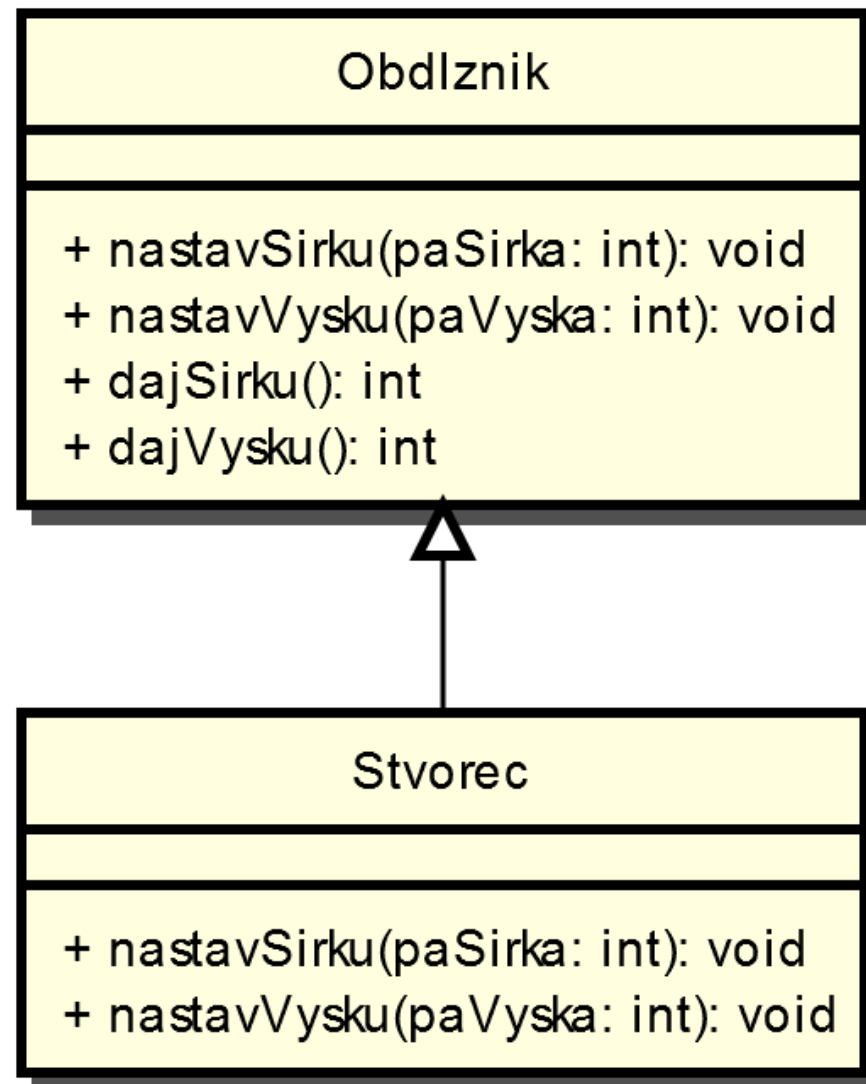
- ak existuje vlastnosť predka, ktorú nemôže potomok splniť = porušený substitučný princíp.
- ľubovoľná referencia na inštanciu predka by mala byť nahraditeľná referenciou na inštanciu potomka bez ovplyvnenia funkcionality.
- vlastnosti – uvedené v dokumentácii

# Dedičnosť: Obdlznik – Stvorec<sub>(1)</sub>



- štvorec – špeciálny prípad obdĺžnika
- obe strany rovnaké

# Dedičnosť: Obdlznik – Stvorec<sub>(2)</sub>



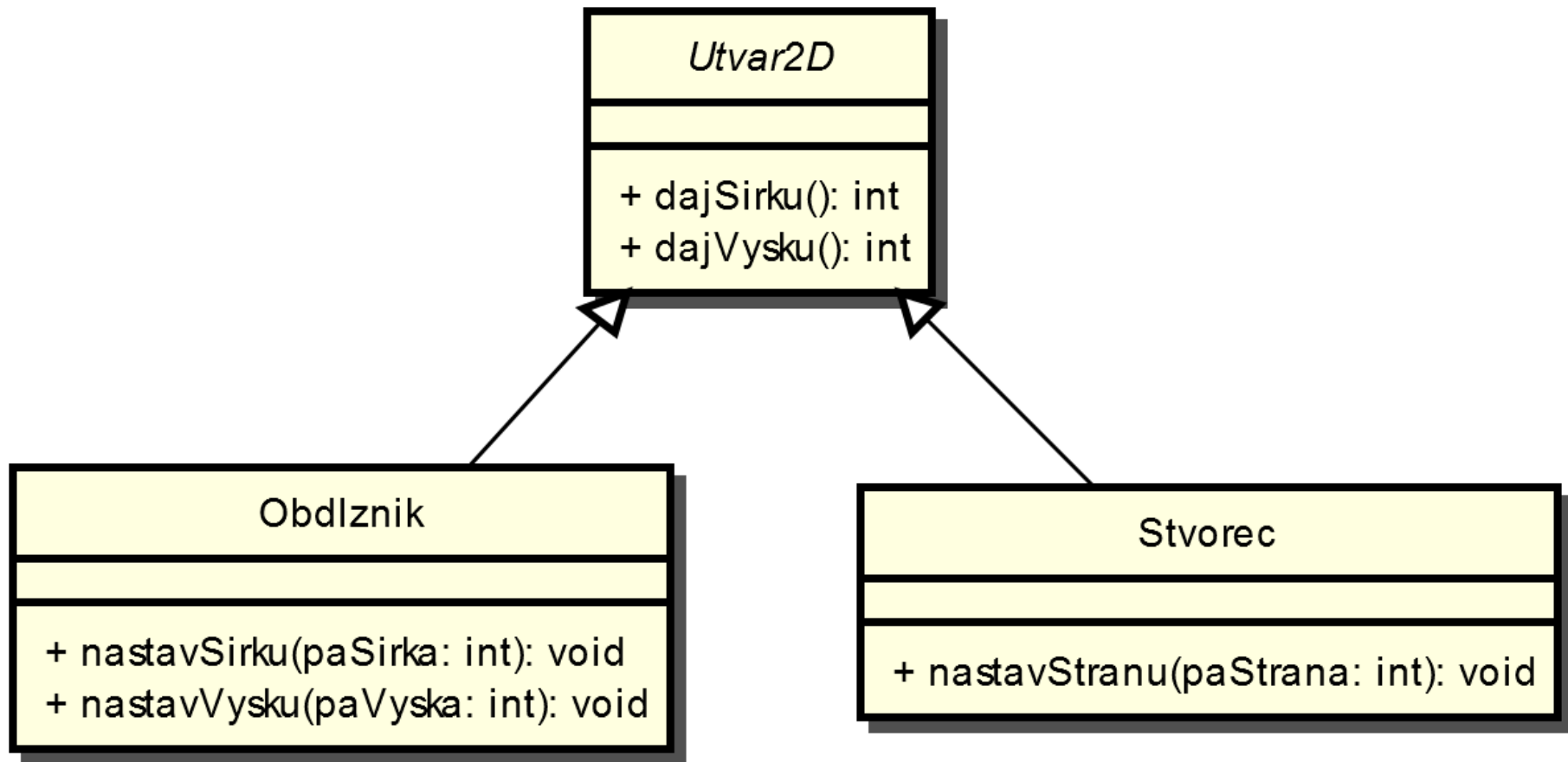
# Metóda nastavSirku v triede Stvorec

```
public void nastavSirku(int paSirka)
{
    super.nastavVysku(paSirka);
    super.nastavSirku(paSirka);
}
```

# Dedičnosť: Obdlznik – Stvorec<sub>(1)</sub>

- obdlžnik a štvorec sú rôzne útvary
  - niektoré vlastnosti rovnaké
  - niektoré vlastnosti rôzne
- 
- rovnaké vlastnosti – abstraktný predok Utvar2D
  - špecifické vlastnosti – konkrétni potomkovia
- 
- polymorfizmus – abstraktné metódy predka implementujú potomkovia svojím spôsobom

# Dedičnosť: Obdlznik – Stvorec<sub>(2)</sub>



# Riziká porušenia LSP

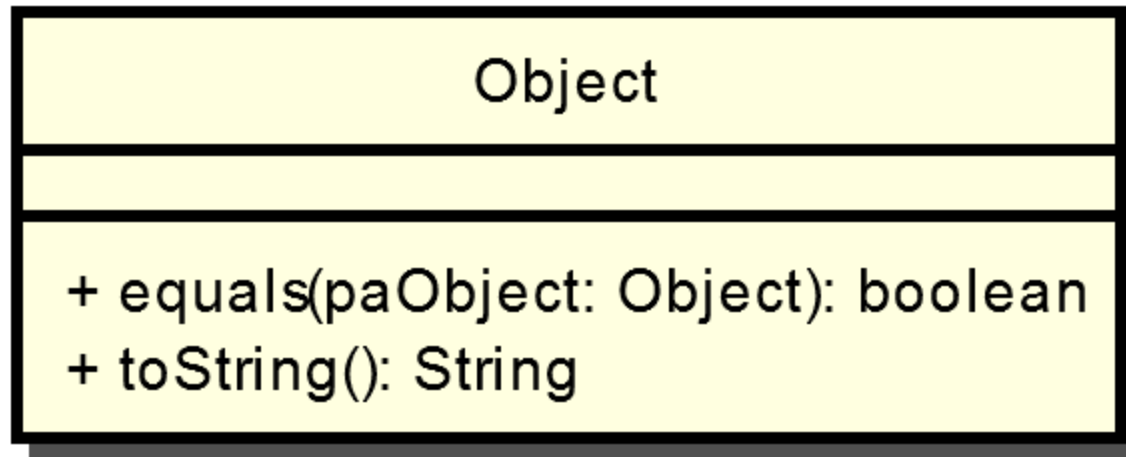
- predok a potomok sú konkrétne triedy
- potomok prekrýva metódy predka

# Znižovanie rizika porušenia LSP

- implementácia abstraktných metód
- zvážiť vzťah dedičnosti konkrétnych tried
- zvážiť prekrývanie konkrétnej metódy predka
- zvážiť využívanie metódy predka pomocou super v prekrývajúcej metóde
  
- zvážiť = konzultovať dokumentáciu predka
  - čítať aj medzi riadkami 😊



# Java – trieda Object



# Object – toString

- textová reprezentácia objektu
- štandardne
  - názov Triedy objektu (dynamický typ)
  - znak@
  - adresa objektu v pamäti
- `System.out.println(objekt)`
- reťazcový operátor `+`
  - objekt nie je reťazec – automatické použitie `toString`

# Trieda Object a LSP

- nie je definovaná ako abstraktná
- má zmysel vytvárať inštancie?
- väčšinou používame ako abstraktnú
- porušenie LSP?

# Trieda Object a LSP – toString

- vlastnosť objektu je definovaná dokumentáciou.
- „Vytvára textovú reprezentáciu inštancie.“
  - dokumentácia štandardnej knižnice Java
- LSP neporušujeme pri takomto chápaní významu metódy toString.

# Trieda Object a LSP – equals

- vlastnosť objektu je definovaná dokumentáciou.
- „Indikuje, či je iný objekt zhodný s týmto“
  - dokumentácia štandardnej knižnice Java
- dokumentácia definuje už spomenuté podmienky
- zachovanie podmienok v dokumentácii = neporušovanie LSP

# Object – equals<sub>(1)</sub>

- relácia ekvivalencie pre dva objekty (referencie)
- štandardne
  - ekvivalentná s operátorom ==
- podmienky relácie:
  - x, y, z: rôzne od null
  - reflexívna: `x.equals(x)` = **true**
  - symetrická: `x.equals(y)`  $\leftrightarrow$  `y.equals(x)`
  - tranzitívna: `x.equals(y)` and `y.equals(z)`  $\leftrightarrow$  `x.equals(z)`
  - prístupová metóda, nemení stav porovnávaných objektov
  - `x.equals(null)` = **false**

# Object – equals<sub>(2)</sub>

- == rovnosť identity dvoch referencií
  - implikuje rovnosť stavov
- equals – rovnosť stavu dvoch objektov
- pri prekrytí musia byť dodržané podmienky

# Dedičnosť – zvyšovanie závislosti

- nutnosť poznania implementácie predka
- =>
- porušenie zapúzdrenia
  - zvyšovanie implementačnej závislosti medzi triedami



# Príklad závislosti – predok<sub>(1)</sub>

```
public class Katalog
{
    private ArrayList<AVIDIelo> aDiela;

    public Katalog()
    {
        aDiela = new ArrayList<AVIDIelo>();
    }

    ...
}
```

# Príklad závislosti – predok<sub>(2)</sub>

```
public void pridaj(AVIDielo paDielo)
{
    aDiela.add(paDielo);
}
```

```
public void pridajZoznam
           (Collection<AVIDielo> paZoznam) {
    for (AVIDielo dielo : paZoznam) {
        this.pridaj(dielo);
    }
}
```

# Príklad závislosti – potomok<sub>(1)</sub>

```
public class PocitaciKatalog extends Katalog
{
    private int aPocet;

    public PocitaciKatalog()
    {
        aPocet = 0;
    }

    ...
}
```

# Príklad závislosti – potomok<sub>(2)</sub>

```
public void pridaj(AVIDielo paDielo)
{
    aPocet++;
    super.pridaj(paDielo);
}
```

```
public void pridajZoznam
           (Collection<AVIDielo> paZoznam) {
    aPocet += paZoznam.size();
    super.pridajZoznam(paZoznam);
}
```

# V čom je problém?

- správne počíta, ak používame správu pridaj()
- ak použijeme pridajZoznam(), ráta zle
  - zaráta dvojnásobok
- kde je chyba?

# Problém

```
public void pridaj(AVIDielo paDielo)
{
    aDiela.add(paDielo);
}
```

```
public void pridajZoznam
           (Collection<AVIDielo> paZoznam) {
    for (AVIDielo dielo : paZoznam) {
        this.pridaj(dielo);
    }
}
```

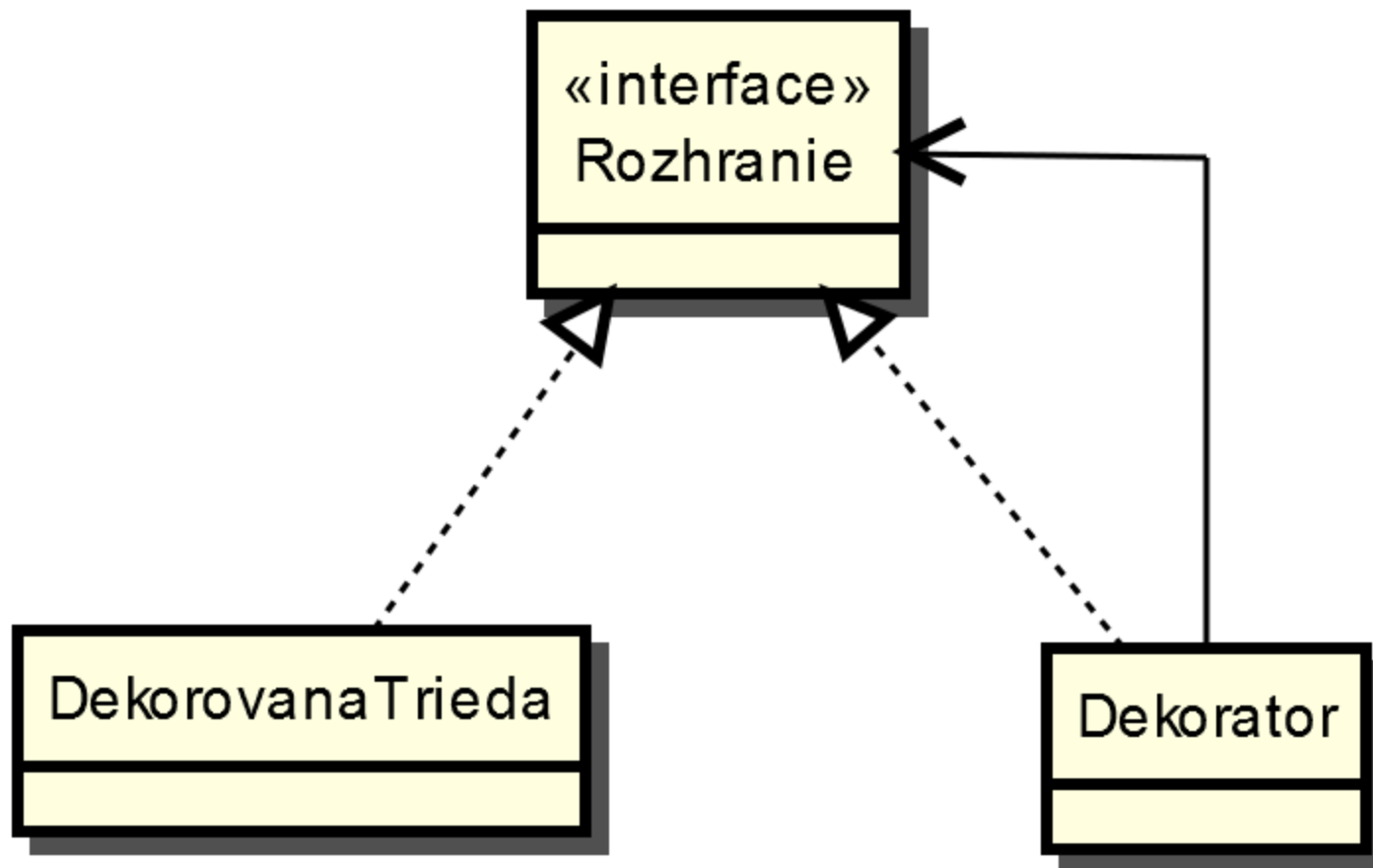
- upraviť predka so znalosťou potomka
  - odstrániť polymorfizmus v pridajZoznam
    - `this.pridaj(dielo)` – problém
    - `aDiela.add(dielo)` – riešenie
- upraviť potomka so znalosťou predka
  - neprekryť metódu pridajZoznam
- odstrániť dedičnosť

# Odstránenie dedičnosti

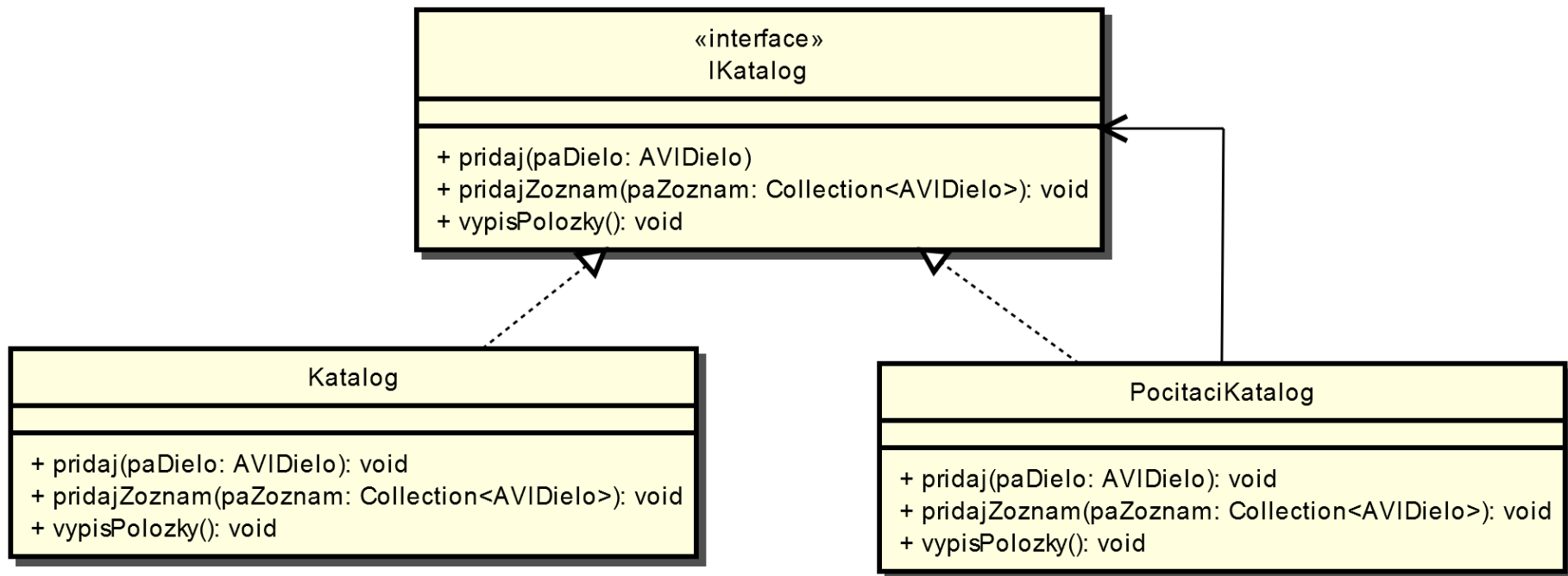
- skladanie
  - už sme skúšali
- zaujímavé rozšírenie – návrhový vzor Dekorátor
  - zabalenie objektu – skladanie
  - delegovanie časti služieb na zabalený objekt



# Návrhový vzor Dekorátor



# Aplikácia na KCalB



# Riešenie cez Dekorátor<sub>(1)</sub>

```
public class PocitaciKatalog implements IKatalog
{
    private int aPocet;
    private IKatalog aZabaleny;

    public PocitaciKatalog(IKatalog paZabaleny) {
        aPocet = 0;
        aZabaleny = paZabaleny;
    }
    ...
}
```

# Riešenie cez Dekorátor<sub>(2)</sub>

```
public void pridaj(AVIDielo paDielo) {  
    aPocet++;  
    aZabaleny.pridaj(paDielo);  
}
```

```
public void pridajZoznam  
            (Collection<AVIDielo> paZoznam) {  
    aPocet += paZoznam.size();  
    aZabaleny.pridajZoznam(paZoznam);  
}
```

# Riešenie cez Dekorátor<sub>(3)</sub>

```
public void vypisPolozky()  
{  
    aZabaleny.vypisPolozky();  
}
```

# Využitie katalógu cez dekorátor

```
IKatalog kat = new Katalog();
```

```
IKatalog kat = new PocitaciKatalog(new Katalog());
```

# Dekorátor – pre a proti

- Výhody
  - nenastáva polymorfizmus cez this
  - triedy nie sú naviazané – ich vzťah možno kedykoľvek meniť
    - minimálna až žiadna závislosť
- Nevýhody
  - nenastáva polymorfizmus cez this
  - chýba reuse rozhrania – skladanie
    - kúsok ukecanejšie

# Polymorfizmus a dedičnosť<sub>(1)</sub>

- polymorfizmus – definuje chovanie objektov
- vychádza zo základného princípu – posielania správ
- nezávisí od dedičnosti
- „čistý“ princíp – len chovanie



# Polymorfizmus a dedičnosť<sub>(2)</sub>

- dedičnosť – definícia hierarchie typov
- definuje štruktúru objektov
- napĺňa princíp reuse
- poskytuje implementačný komfort
- zahŕňa aj polymorfizmus
  - prekrývanie metód
  - implementácia abstraktných metód
- „zmiešaný“ princíp – štruktúra + chovanie

# Názov „dedičnosť“

- dedičnosť dovoľuje zaviesť hierarchiu typov
- dedičnosť – nesprávny názov
  - neznamená dedenie génov
  - neznamená dedenie majetku
- vzťah is-a, generalizácia/špecializácia
- odvádza pozornosť od podstaty – hierarchia typov – k implementačným detailom – čo trieda dedí
- UML – pojem Generalizácia

# Vďaka za pozornosť