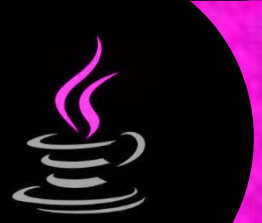
Znovupoužiteľnosť

1. 12. 2011
ÚINF/PAZ1c
(Róbert Novotný)

Znovupoužiteľnosť!

- nebudeme vynachádzať ni teplú vodu, ni koleso, ni triedenie, ni spojový zoznam
- dobre navrhnutá trieda sa dá ukradnúť znovupoužiť do iného projektu
- vznikajú tak knižnice tried, kde megaprojekt vystaviame zo znovupoužiteľných súčastí



2 (3) spôsoby výstavby tried

kompozícia

stav zložitej triedy je tvorený jednoduchšími triedami

odvodzovanie / špecializácia z jednoduchšej / všeobecnejšej triedy

dedičnosť

hybrid kompozície a dedičnosti

delegácia

Dedičnosť (inheritance) tried

- umožňuje **odvodiť** objekt od iného
  - pridať mu nové schopnosti / stav
  - pozmeniť existujúce schopnosti

Auto **je vozidlo**, ktoré je motorové, dvojstopové a poháňané benzínom.

Študent **je človek** navštevujúci školu v danom ročníku.

Človek **je dvojnožec**, ktorý nemá perie.

-- Platón, Politikus, 4. stor. pnl

Kedy kompozícia a kedy dedičnosť?

- množstvo hádok a debát
- niekedy je hranica nejasná
- niekedy zistíme až po čase, že sme zvolili zle
- niekedy sa dá jeden postup nahradiť druhým

Uprednostňujte kompozíciu pred dedičnosťou!

-- Gang of Four, autori Design Patterns

Poznámky k dedičnosti a kompozícii

- pôvodné triedy sú často dodávané ako binárky
  - niekto nám ich venuje, predá..
- nemáme ich zdrojový kód a teda ich nemôžeme meniť'

Vďaka kompozícii a dedičnosti ich nemusíme meniť!

Kedy kompozícia a kedy dedičnosť?

Kompozícia	Dedičnosť
Has-a?	Is-a?
Objekt má / pozostáva z / je tvorený z iného objektu	Objekt je špeciálnym prípadom / odvodený z iného objektu
Používateľ má priateľov. Študijný program má predmety.	Pleso je jazero, ktoré... • Každé pleso je jazerom. Žirafa je zviera, ktoré... • Každá žirafa je zvierať om.

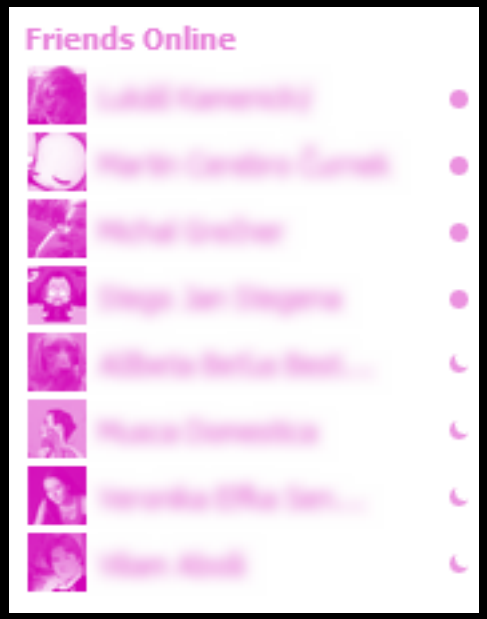
Kedy kompozícia a kedy dedičnosť?

- vytvorené vety by mali dávať **zmysel**
- ak sú „**čudné**" alebo „**ak by sme sa na to pozreli *takto*, tak to dáva zmysel**", treba sa zamyslieť nad správnym použitím.

Auto extends Motor	Každé auto je motorom.	Nedáva zmysel.
Auto has-a Motor	Každé auto má motor.	Dáva zmysel.
Žirafa extends Zviera	Každá žirafa je zvieraťom.	Dáva zmysel.
Žirafa has-a Zviera	Žirafa má zviera.	Nedáva zmysel.

Príklad: zoznam kontaktov

- vyvíjame *yet-another* sociálnu sieť
- používateľ chce mať **zoznam priateľov**
- ako ho implementovať?



Príklad: zoznam kontaktov

- zásada: zistime funkcionalitu
- čo očakávame od zoznamu?
  - pridaj kontakt
  - odober kontakt
  - zisti, či je kontakt online

```
class Kontakt {
  int id
  String nick
}
```

```
interface Kontakty {
  void pridaj(Kontakt k);
  void odober(Kontakt k);
  boolean jeOnline(Kontakt k);
}
```

Diskusia
Koho zodpovednosťou je vedieť, či je kontakt online?

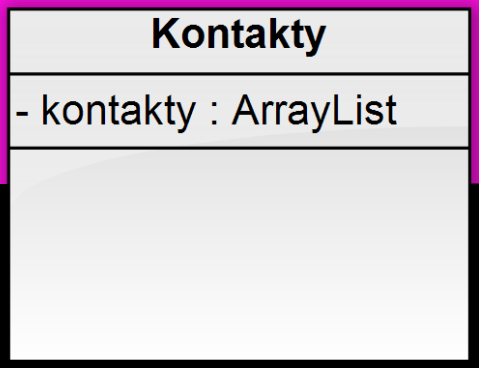
Riešenie č. 1 [kompozícia]

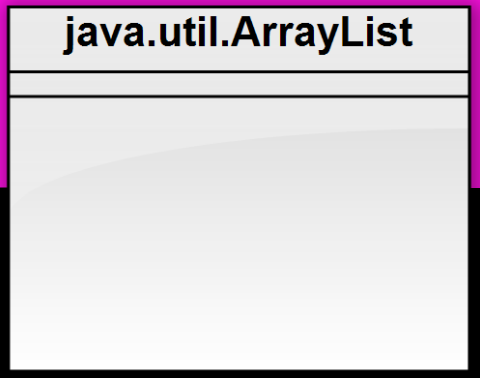
- využijeme **kompozíciu** a štandardný zoznam **ArrayList**

```
public class Kontakty {

  private List<Kontakt> kontakty
    = new ArrayList<Kontakt>();

  /* ... */
}
```

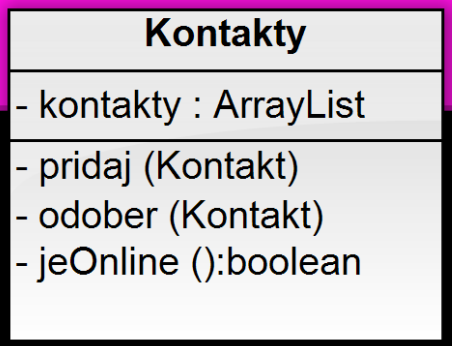


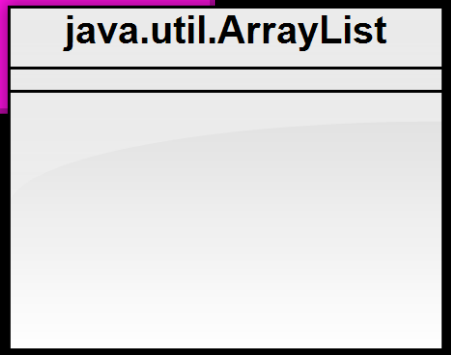


Riešenie č. 1 [kompozícia]

- využijeme **kompozíciu** a štandardný zoznam ArrayList
- metódy budú využívať metódy ArrayListu

```
public void pridaj(Kontakt kontakt) {
   this.kontakty.add(kontakt);
}
```





Príklad: zoznam kontaktov

- zoznam kontaktov je **zoznam**
  - teda kolekcia prvkov
- v Jave máme **java.util.List**
  - interfejs pre zoznamy
  - typické operácie
    - add()
    - remove()
- môžeme sa od neho odpichnúť!

Nástrel interfejsu

```
interface Kontakty
  extends java.util.List<Kontakt>
{
  void pridaj(Kontakt k);
  void odober(Kontakt k);
  boolean jeOnline(Kontakt k);
}
```

- **pridaj()** a **odober()** majú analógie v **List**-e
  - add() a remove()

Nástrel interfejsu

- zatiaľ máme len interfejsy
  - teda dohodnutý kontrakt
  - teda zoznam metód
- **potrebujeme implementáciu** metód
- potrebujeme kód pre všetky **add()**, **remove()**...
- spolu **42** metód!

Využime hotové veci!

- Načo vymýšľať niečo, čo je hotové?
- Využime existujúcu triedu!

java.util.AbstractList

- Oddedíme a **prekryjeme** požadované metódy
- Dokumentácia káže prekryť metódy
  - **get(int)**, **size()**, **set(int, E)**, **add(int, E)** a **remove(int)**
  - nezabudnime ešte prekryť **jeOnline()**

Čo získame?

- získame triedu **Kontakty**
  - vieme robiť všetko, čo sme si na začiatku nadefinovali
- trieda **Kontakty** sa správa ako **List**
- vieme ju použiť všade tam, kde sa očakáva List
- napr. vieme triediť kontakty pomocou **Collections.sort()**

Čo sme stratili?

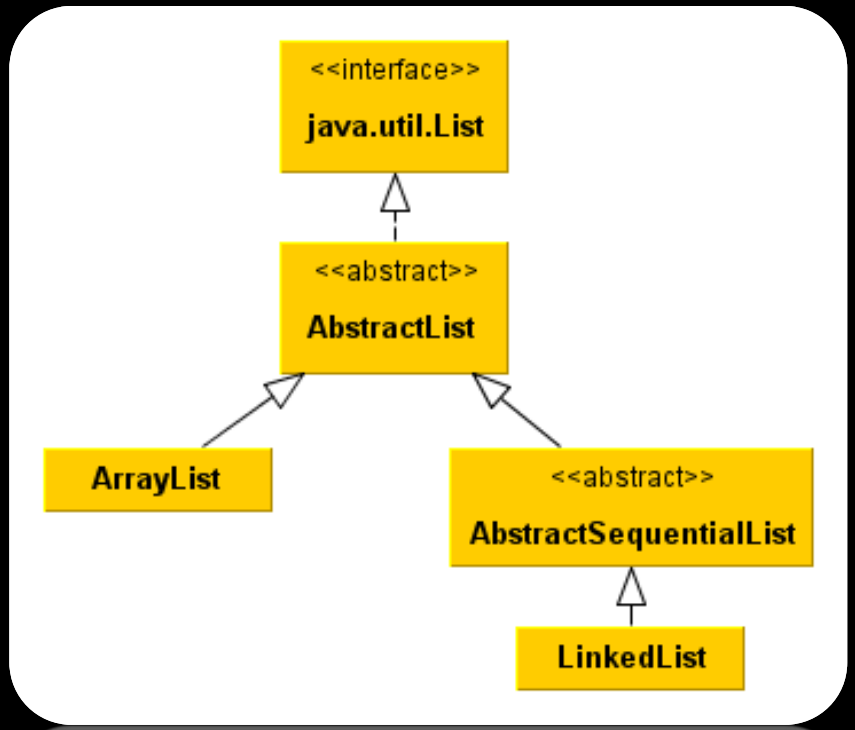
- dedíme od triedy AbstractList
- v Jave: **neexistuje viacnásobná dedičnosť!**
- ak náhodou naše kontakty potrebujú dediť od inej triedy, máme problém!

Riešenie:
od jednej triedy oddedíme, v druhej implementujeme interfejs a použijeme delegáciu

o tom neskôr

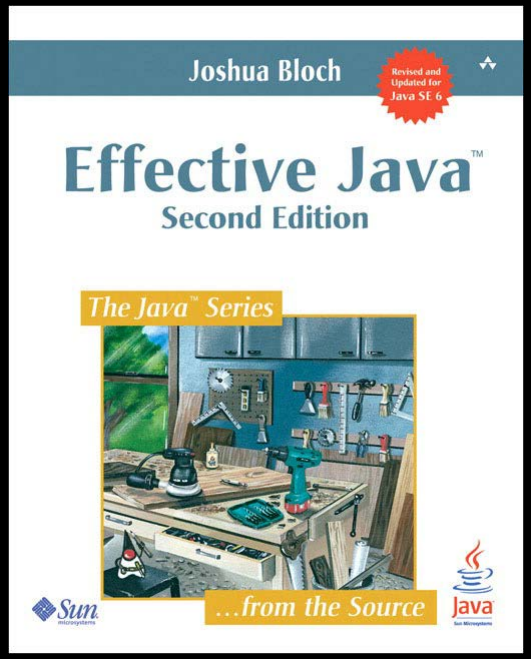
Hierarchia tried zoznamov

- ak dedíme, **ktorú** triedu si **vybrať**?
- dostatočne **všeobecnú**, ale nie príliš **konkrétnu**
- obvykle pomôže len **dokumentácia**!



Kolekcie a ich tragédia

- Joshua Bloch – Effective Java, 2. vydanie (2008)
  - krátka, ale geniálna kniha o zákutiach Javy
- príklad:
  - urobme množinu, ktorá si bude evidovať počet pridaných prvkov
  - „**do tejto množiny bolo pridaných X prvkov**"



Návrh kódu a logika uvažovania

- oddedíme z **HashSet**
- prekryjeme metódu **add()**
  - pripočítame jednotku
  - zavoláme rodičovskú metódu, ktorá pridá prvok
- musíme prekryť aj **addAll()** (pridanie kolekcie do množiny)
  - pripočítame toľko, koľko je prvkov v množine
  - zavoláme rodičovskú metódu, ktorá pridá prvky

Návrh kódu a logika uvažovania

```
public class InstrumentedHashSet<E> extends HashSet<E> {

  private int početPridaní = 0;

  public boolean add(E e) {
    početPridaní++;
    return super.add(e);
  }

  public boolean addAll(Collection<? extends E> c) {
    početPridaní += c.size();
    return super.addAll(c);
  }
}
```

Použitie triedy

```
Set<String> s = new InstrumentedHashSet<String>();
s.addAll(Arrays.asList("Baldrick", "Edmund", "Queenie"));
```

- lenže ak zistíme počet pridaných prvkov, zistíme, že máme výsledok **6**
- **prečo?** nik nevie
- pozrieme do zdrojákov!
  - ešteže ich Oracle zverejňuje...
- vinník: metóda **addAll()** v java.util.**AbstractCollection**

Návrh kódu a logika uvažovania

```
public boolean addAll(Collection<? extends E> c) {
  boolean modified = false;
  Iterator<? extends E> e = c.iterator();
  while (e.hasNext()) {
    if (add(e.next())) modified = true;
  }
  return modified;
}
```

Metóda **addAll()** volá metódu **add()**! Hm!

- započíta sa to dvakrát – raz v prekrytej metóde **addAll()**, ktorá zavolá našu prekrytú metódu **add()**

Čuduj sa svete, dokumentácia!

addAll

```
public boolean addAll(Collection c)
```

Adds all of the elements in the specified collection to this collection (optional operation). The behavior of this operation is undefined if the specified collection is modified while the operation is in progress. (This implies that the behavior of this call is undefined if the specified collection is this collection, and this collection is nonempty.)

This implementation iterates over the specified collection, and adds each object returned by the iterator to this collection, in turn.

Note that this implementation will throw an `UnsupportedOperationException` unless `add` is overridden (assuming the specified collection is non-empty).

Specified by:

`addAll` in interface `Collection`

Parameters:

`c` - collection whose elements are to be added to this col

Returns:

`true` if this collection changed as a result of the call.

Throws:

`UnsupportedOperationException` - if this collection
`addAll` method.

`NullPointerException` - if the specified collection is null.

See Also:

`add(Object)`

Z dokumentácie možno odvodiť toto správanie.

Čo ak dokumentácia nie je?

Dedičnosť je bezpečná, ak...

- dedíte od tried v rovnakom balíčku
- máte dosah na implementáciu rodičovských tried
  - vidíme ich zdrojáky a vieme ich upraviť
- dedíte od tried, ktoré boli explicitne navrhnuté na dedenie a hovoria o tom v dokumentácii
- implementujete interfejs alebo tvoríte interfejs, ktorý dedí od iného interfejsu

Dedičnosť je bezpečná, ak...

- príklad s HashSetom nefunguje
- ale príklad so počítajúcim zoznamom by fungoval
  - LinkedList/ArrayList nepoužívajú v addAll() metódu add()

Dediť od náhodných tried krížom cez balíčky môže viesť k nečakaným problémom!

Riešenie problému

- **zrušiť pripočítavanie** v metóde addAll()
  - lebo sme si prečítali dokumentáciu
  - lenže my sa spoliehame na implementačný detail, ktorý sa môže zmeniť, a potom máme problém

```
public boolean addAll(Collection<? extends E> c) {
  početPridaní += c.size();
  return super.addAll(c);
}
```

Riešenie problému č. 2

- prekryť si metódu addAll() po svojom
  - prelez kolekciu, pridaj prvok po svojom bez metódy **add()**
  - lenže časom môžeme zistiť, že kopírujeme kód z metódy rodičovskej triedy

```
Iterator<? extends E> e = c.iterator();
while (e.hasNext()) {
  if (pridajPoSvojom(e.next())) modified = true;
}
return modified;
```

Riešenie problému: delegácia!

```
public class InstrumentedSet<E> implements Set<E> {
  private final Set<E> s;

  public InstrumentedSet(Set<E> s) { this.s = s; }

  public void clear() { s.clear(); }

  public boolean contains(Object o) {
    return s.contains(o);
  }
  public boolean isEmpty() {
    return s.isEmpty();
  }
  // .. ďalšie metódy
}
```

objekt, na ktorý budeme delegovať volania

v metóde delegujeme volanie na príslušný objekt

Nástrel delegácie - pokračovanie

```
public class InstrumentedSet<E> implements Set<E> {
  // private final Set<E> s;
  // pokračovanie z predošlého snímku

  private int početPridaní = 0;

  public boolean add(E e) {
    početPridaní++;
    return s.add(e);
  }

  public boolean addAll(Collection<? extends E> c) {
    početPridaní += c.size();
    return s.addAll(c);
  }
}
```

V metóde delegujeme volanie na príslušný objekt
Je to analógia **super**.

Použitie triedy

```
Set<String> set
    = new InstrumentedSet<String>(new HashSet<String>());
```

- množina bude delegovať volania na objekt typu **HashSet**
- množina **set** sa správa v metódach **add()** a **addAll()** podľa nášho želania, ostatné metódy sú delegované na **HashSet** v parametri konštruktora

Výhody a nevýhody delegácie

- simulujeme dedičnosť
- nemusíme sa spoliehať na neznáme správanie rodičovskej triedy
- môžeme dodať funkcionalitu pre počítanie pridaných prvkov do ľubovoľnej triedy, ktorá implementuje **Set**
- nevýhody:
  - treba vygenerovať kód pre delegovanie
    - rozumné IDE to spraví za nás
  - kód vyzerá na pohľad nepríjemne, ale je len zjavný

Rada starej matere

- dedičnosť narúša zapúzdrenie
  - dedičnosť **vyžaduje** znalosti o správaní rodičovskej triedy

Riešenie
Uprednostňujte kompozíciu pred dedičnosťou

INTERFEJSY, KONTRAKTY A DEDIČNOSŤ

Interfejsy a spomienky

- interfejs = kontrakt
  - ja som vec, ktorá ti poskytne robiť...ak...
  - hovorí ČO daný objekt poskytuje
- v implementácii sa povie AKO sa to dosiahne

Zásada pre používanie interfejsov

Programujte vzhľadom k interfejsom, nie k ich implementáciám

Program to an interface, not to an implementation!

—Gang of Four (Gamma, Helm, Johnson, Vlissides), **Design Patterns**

Programujte vzhľadom k rozhraniam!

- vždy, keď je to možné, používajte pre premenné interfejsy, nie konkrétne implementácie
  - v parametroch metód
  - v návratových hodnotách
  - v lokálnych premenných na ľavej strane
- umožníte tým dodržať zásadu čiernej skrinky

Interfejsy a implementácie

```
public class SprávcaŠtudentov {
  private List<String> študenti = new ArrayList<String>();
  public List<String> getŠtudenti() {
    return študenti;
  }
}
```

- **interfejsy** v type inštančnej premennej
  - klienta zaujíma, že dostane zoznam študentov, nemusí ho zaujímať, ako je tento zoznam implementovaný (tu: **ArrayList** = zoznam nad poľom)
- **implementácie** pri konštruovaní

Akú to má výhodu?

- ak všade používame implementácie a rozhodneme sa nahradiť ich, máme problém
  - všade používajme **ArrayListy**
  - lenže zrazu sa ukáže, že ich chceme nahradiť **CopyOnWriteArrayListom**, ktorý podporuje súčasný prístup z viacerých vlákien

Akú to má výhodu?

- zbesile nahrádzanie v kóde
- **strašné** narušenie kompatibility
  - kód, ktorý používa naše triedy sa musí prepísať
  - „pokazia" sa oddedené triedy

Reálny príklad (pozitívny)

```
public setModel(ListModel model)…
```

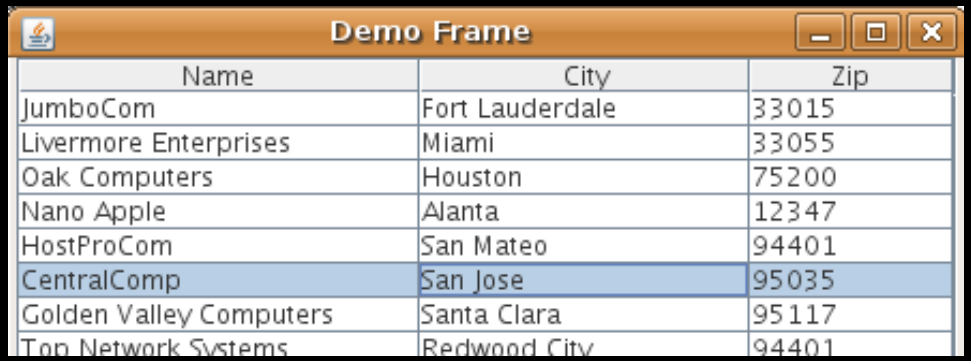
- trieda **JList**: zobrazenie zoznamov
- javax.swing.**ListModel** – model (nositeľ) dát zobrazených v zozname
- môžeme dodať štandardný model **DefaultListModel**
  - **ListModel** simulujúci zoznam
- alebo môžeme dodať ľubovoľnú inú vlastnú implementáciu
  - ťaháme dáta z databázy, súborového systému...

Reálny príklad (aj majster tesár...)

- trieda **JTable**: zobrazenie tabuľkových dát

```
public JTable(Vector rowData, Vector columnNames)…
```

- java.util.**Vector** – trieda pre zoznamy (ťa ArrayList)
- uvedenie implementačnej triedy robí z tohto konštruktora **nepoužiteľnú** (= zbytočnú) vec



Kdepak udělali soudruzi zo Sunu chybu?

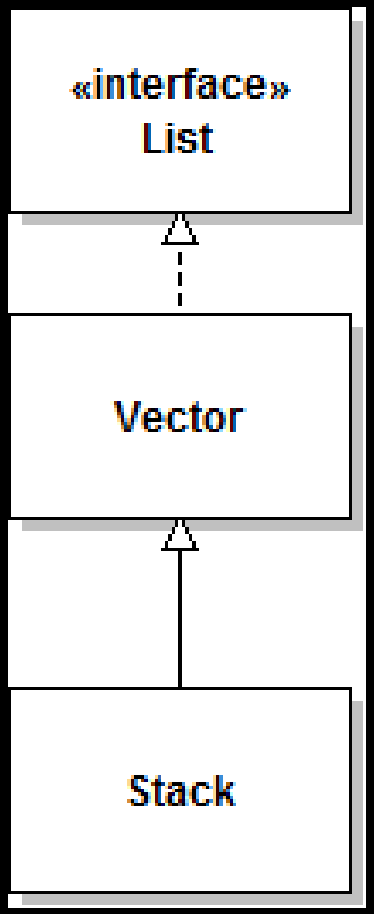
- Java 1.0: jediná trieda pre zoznamy: java.util.**Vector**
- Java 1.2 (1998): Joshua Bloch prekopal triedy pre kolekcie
  - kopa interfejsov (**List**, **Set**, **Map**)...
- **Vector** sa stal zastaralým
  - namiesto neho **ArrayList**
  - ešteže ArrayList implements List
  - ba dokonca Vector implements List

Kdepak udělali soudruzi zo Sunu chybu?

- žiaľ, konštruktor v JTable je nepoužiteľný, lebo očakáva implementáciu a nie interfejs
- ospravedlnenie: vtedy sa mnoho vecí nevedelo predvídať
- Milý Oracle, mohli by ste dodať...

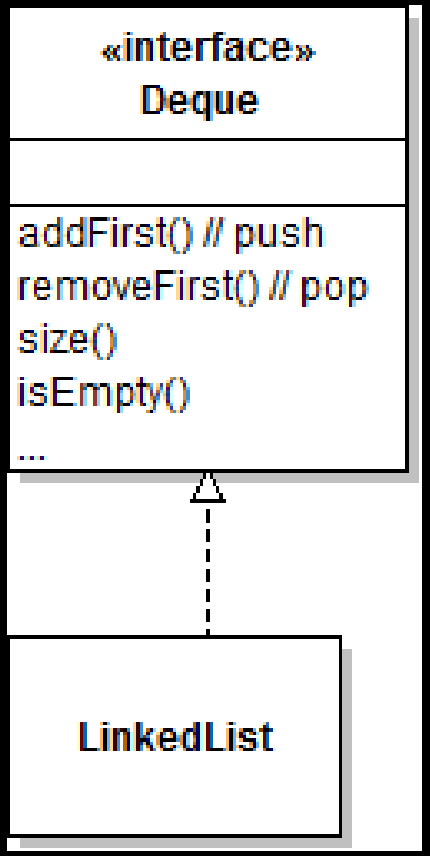
```
public JTable(List rowData, List columnNames)…
```

Iný príklad utnutého majstra



- **Stack extends Vector**
- narúša to pravidlo „je"
  - „zásobník nie je vektorom založeným na poli"
- narúša to Liskovovej substitučný princíp
  - zásobník je zoznam, do ktorého možno vkladať len na koniec

Oprava chýb minulosti



- v JDK 6 – **Deque**
- interfejs pre „double-ended queue"
  - rad s dvoma koncami
- existuje viacero implementácií
  - **LinkedList**
  - **ArrayDeque**

Otázky?

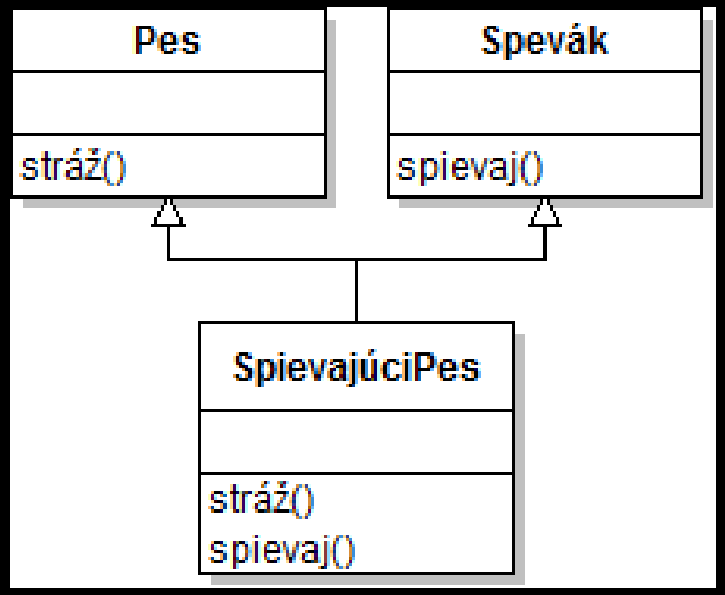
Interfejs môže naznačovať rolu

- Príklad: máme triedu **Pes** a triedu **Spevák**. Chceme **spievajúceho psa**
  - **Pes** – trieda poskytujúca schopnosť strážiť
  - **Spevák** – trieda poskytujúca schopnosť lúdiť tóny

```
class Pes {
   void stráž() {
      …
   }
}
```

```
class Spevák {
   void spievaj() {
      …
   }
}
```

Viacnásobná dedičnosť

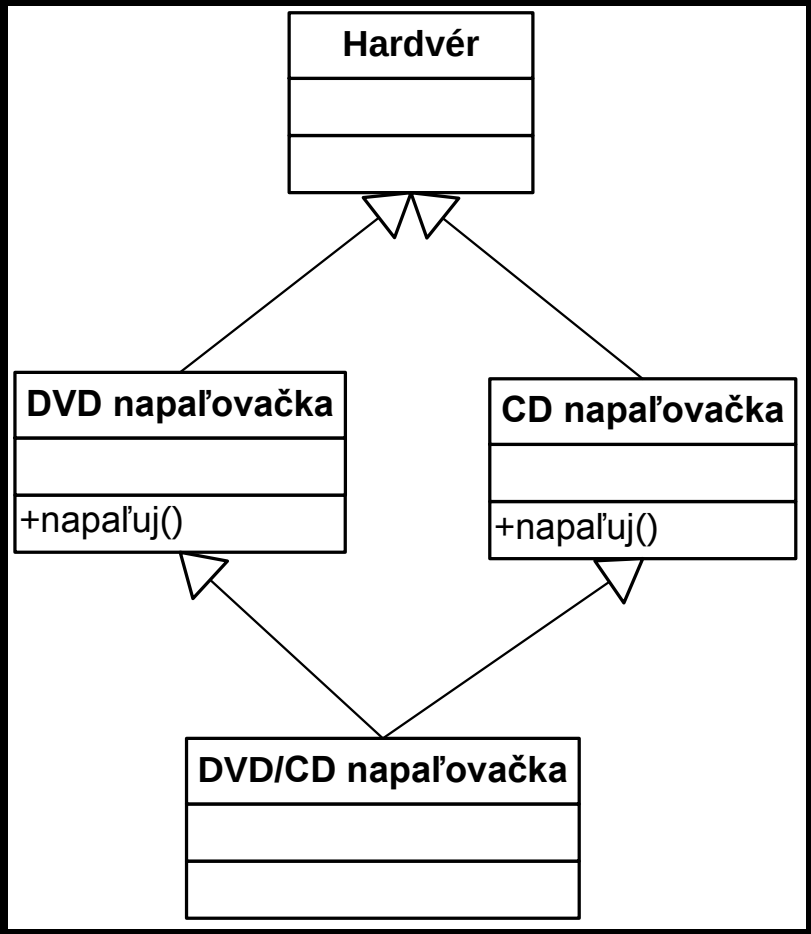


- viacnásobná dedičnosť však v Jave neexistuje!
- nemožno dediť od viacerých tried

```
class SpievajúciPes
   extends Pes, Spevák {
...
}
```

Viacnásobná dedičnosť bola explicitne zakázaná

- „Smrtiaci smaragd smrti" (Deadly diamond of Death)
- DVD/CD napaľovačka zdedí obe metódy na napaľovanie – ale ktorá sa má zavolať kedy?



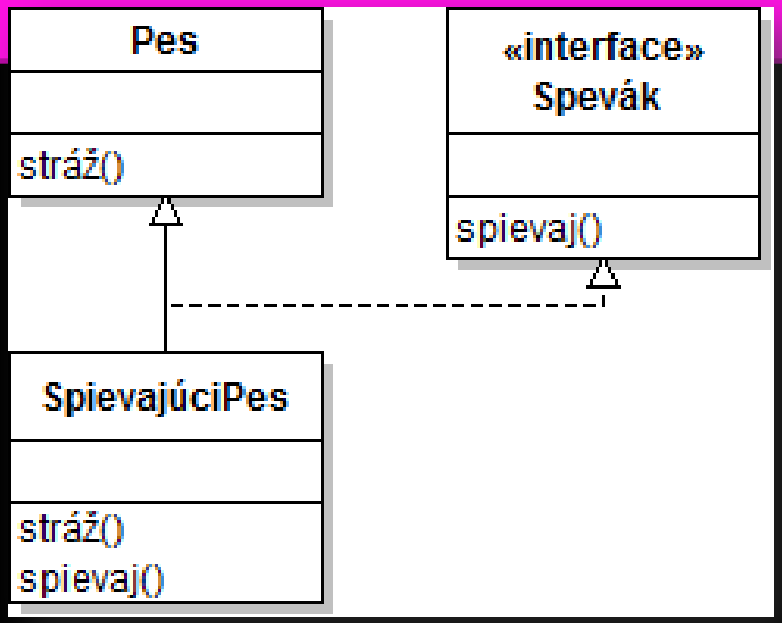
Nie je to však tragédia

- snaha o viacnásobnú dedičnosť často znamená snahu o triedu, ktorá napĺňa viacero rolí
- **spievajúci pes** – **pes**, ktorý dokáže naplniť rolu speváka
- roly vyjadríme interfejsom
- trieda naplní rolu tým, že implementuje interfejs

```
interface Spevák {
  void spievaj();
}
```

Implementácia interfejsu

```
public SpievajúciPes extends Pes implements Spevák {
   public void spievaj() {
       System.out.println("Máááám rozpráávkovú búúúdu!");
   }
}
```



Trieda môže napĺňať viacero rolí

PAZ1C

```
public ŠaľenyPes extends Pes
   implements Spevák, Comparable, Serializable, Cloneable
{
   //..
   //milión metód vyžadovaných od rozhraní
   //...
}
```

```
ŠaľenyPes einstein = new ŠaľenyPes();
Pes p = einstein;

Comparable c = new ŠaľenyPes();
Spevák spevák = new ŠaľenyPes();
```

- funguje dedičnosť aj polymorfizmus

Praktický príklad

- vezmime si typické operácie nad zoznamami
  - vykonaj nad každým prvkom operáciu
  - nájdi (jeden) prvok spĺňajúci kritérium
  - nájdi všetky prvky spĺňajúce kritériá
  - prefiltruj zoznam

Metóda vráti podreťazec od počiatočného indexu po koncový (okrem neho).

Interfejsy môžu nahradiť lepenie kódu

- ako vyzerá algoritmus pre **each**?

1. bež v cykle cez zoznam
2. vezmi prvok
3. *aplikuj naň požadovanú operáciu*
   - ak rátame dvojnásobky, operácia zdvojnásobí prvok
   - ak rátame piate odmocniny, dodáme príslušnú

prvé dva kroky sa nikdy nemenia
tretí rok je premenlivý => vyextrahujeme ho do interfejsu

Prípad: each

- operáciu vieme reprezentovať interfejsom

```
interface Operácia {
  void vykonaj(int element);
}
```

```
public class OperácieSoZoznamami {
  public static void spracuj(List<Integer> list,
                             Operácia o) {
    for(int element : list) {
       o.vykonaj(element);
    }
  }
}
```

Prípad: each – vypíš dvojnásobky

```
public class VypíšDvojnásobky implements Operácia {
  public void vykonaj(int element) {
    System.out.println(element * 2);
  }
}
```

```
List<Integer> cisla = Arrays.asList(2, 4, 6);
Operácia o = new VypíšDvojnásobky();
OperácieSoZoznamami.spracuj(cisla, o)
```

- operáciu, ktorú vykonávame, určíme používaným interfejsom
- stačí vytvoriť triedu implementujúcu interfejs a dodať do nej kód
- metóde **spracuj()** dodáme zoznam a triedu s kódom, ktorý sa má vyvolať pri prechádzaní zoznamu

Interfejsy môžu nahradiť lepenie kódu

- návrhový vzor (design pattern!)
- algoritmus má viacero krokov
- niektoré sú nemenné
- niektoré sa menia a to za behu
- riadky, ktoré používať „maže a mení" vieme vložiť do metódy interfejsu
- v kóde algoritmu potom voláme príslušnú metódu interfejsu, za ktorou je konkrétna implementácia

Interfejsy môžu nahradiť lepenie kódu

- tento návrhový vzor sa používa kade-tade

```
File mp3Adresar = new File("D:/mp3");
File[] súbory = mp3Adresar.listFiles(FileFilter filter);
```

- **FileFilter** umožňuje vložiť inštanciu filtra, ktorý pre každého potomka adresára povie, či sa má vložiť do výsledného poľa.

```
interface FileFilter {
  boolean accept(File file);
}
```

Používateľ musí vytvoriť implementáciu tohto nterfejsu, vytvoriť jej inštanciu a hodiť ju do parametra.

Interfejsy môžu nahradiť lepenie kódu

```
public class Mp3FileFilter implements FileFilter {
  boolean accept(File file) {
     return file.getName().endsWith(".mp3")
  }
}
```

```
File mp3Adresar = new File("D:/mp3");
FileFilter filter = new Mp3FileFilter();
File[] súbory = mp3Adresar.listFiles(filter);
```

- v poli súbory sa zjavia len súbory končiace sa na .mp3