Programovanie, algoritmy, zložitosť / ÚINF/PAZ1c

PAZ1C

Róbert Novotný
robert.novotny@upjs.sk

13. 10. 2010

Ako nevynájsť koleso nanovo

- znovupoužiteľnosť kódu je jedným z cieľov OOP
  - nebudeme vynachádzať ni teplú vodu, ni koleso, ni triedenie, ni spojový zoznam
- dobre navrhnutú triedu možno prevziať a použiť bez zmien v iných projektoch
- vznikajú tak knižnice tried, kde megaprojekt vystaviame zo znovupoužiteľných súčastí

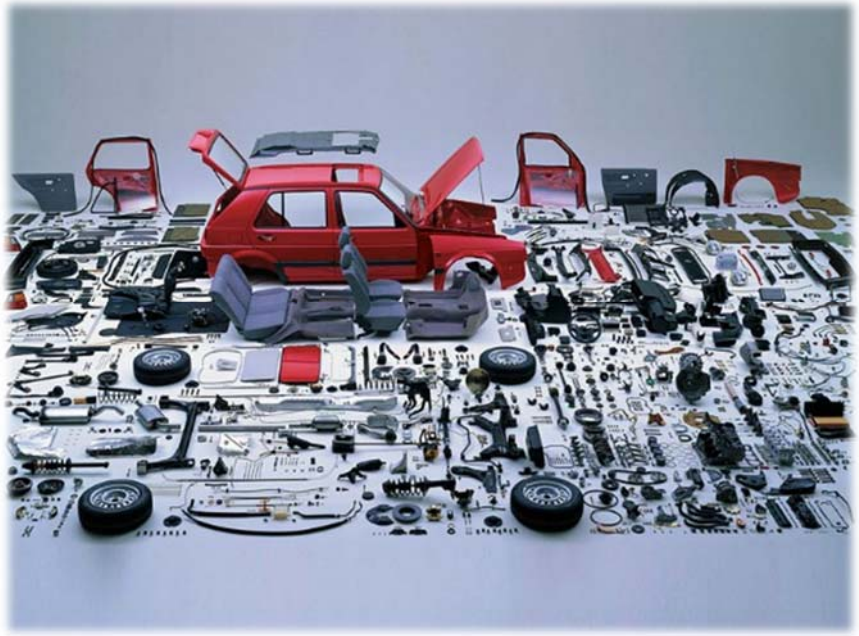
Ako vystavať babylonskú vežu

- v OOP existujú dva (tri) spôsoby ako vystavať zložitú triedu z jednoduchších
- **kompozícia**
  - stav zložitej triedy je tvorený jednoduchšími triedami
- **dedičnosť**
  - odvodzovanie/špecializácia z jednoduchšej/všeobecnejšej triedy

- **delegácia**
  - hybrid

Kompozícia tried

- automobil **sa skladá z** motora, prevodovky, volantu
  - motor **pozostáva z** valcov, sviečok
    - valec **pozostáva z** ...



Kompozícia v OOP

PAZ1C

Kompozícia v OOP je jednoduchá. Súčasti uvedieme ako stav triedy:

```
class Auto {
   Motor motor;
   Prevodovka prevodovka;
   Volant volant;
}
```

```
public class Motor {
  Valec[] valce;
  Sviečky[] sviečky
}
```

```
public class Valec {
   ...
}
```

V ľudskej reči je kompozícia vyjadrená slovom „má“ (*has a*). Auto **má** motor. Motor **má** valec.

Kompozícia v OOP

Inštanciu vystaviame z iných inštancií

```
class Osoba {
  String meno
  Adresa bydlisko
}
```

```
class Adresa {
  String ulica
  int čísloDomu
  String mesto;
  int psč
}
```

```
Adresa adresa = new Adresa();
adresa.setUlica("Hlavná");
adresa.setČísloDomu(25);
adresa.setMesto("Košice");
adresa.setPsč(04001);
```

```
Osoba osoba = new Osoba();
osoba.setMeno("Ján Zlý");
osoba.setBydlisko(adresa);
```

Dedičnosť (inheritance) tried

- dedičnosť 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 sa ukáže, že jeden zvolený postup je ťažkopádnejší než druhý

Uprednostňujte kompozíciu pred dedičnosťou!

-- Gang of Four, autori Design Patterns

- niekedy sa dá jeden postup nahradiť druhým

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 zvieraťom.

Kedy kompozícia a kedy dedičnosť?

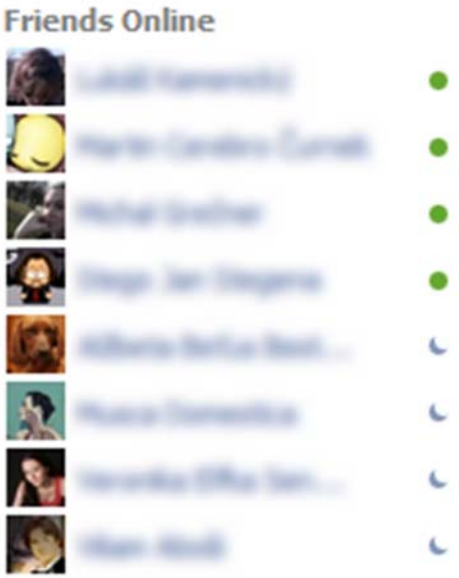
PAZ1C

- 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]

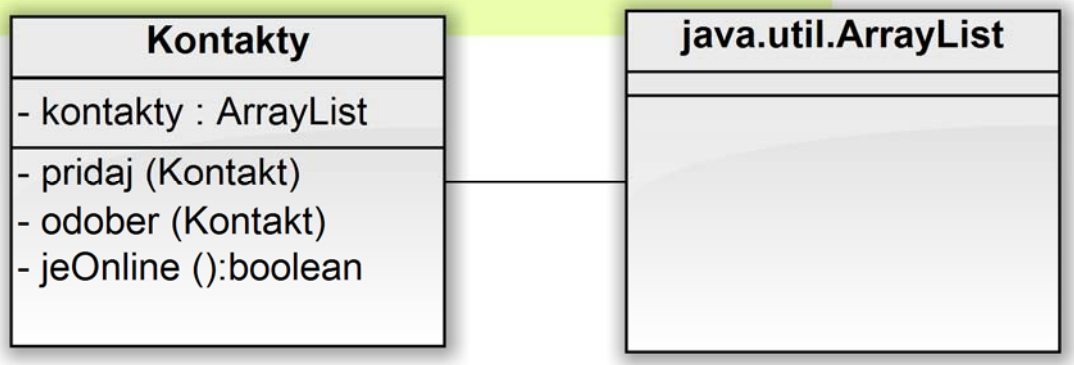
- využijeme **kompozíciu** a štandardný zoznam **ArrayList**

```
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()**...

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?

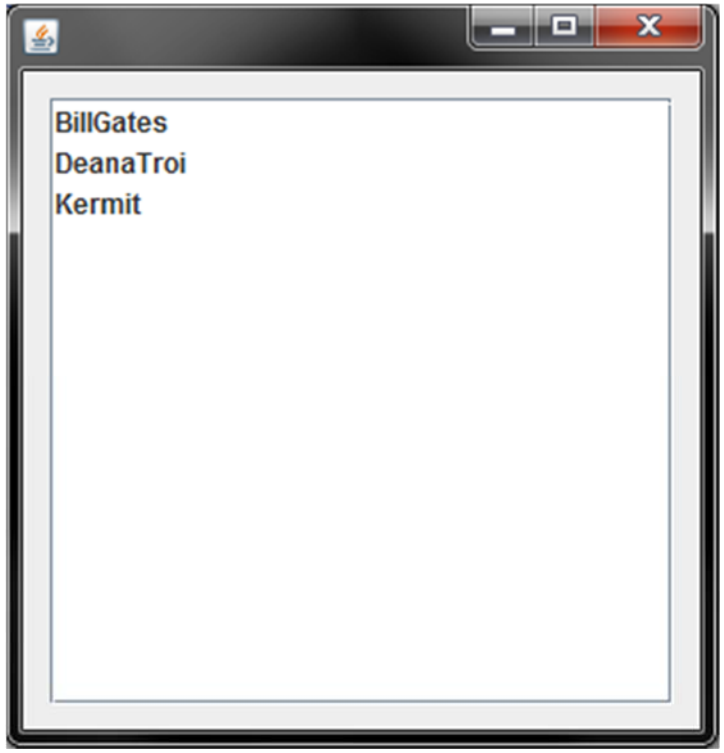
- získame triedu **Kontakty**
  - vieme robiť všetko, čo sme si na začiatku nadefinovali
- trieda **Kontakty** sa správa ako **List**
- vieme ju použiť všade tam, kde sa očakáva List
- napr. vieme triediť kontakty pomocou **Collections.sort()**

Príklad 2

JLIST A ZOBRAZOVANIE PRVKOV

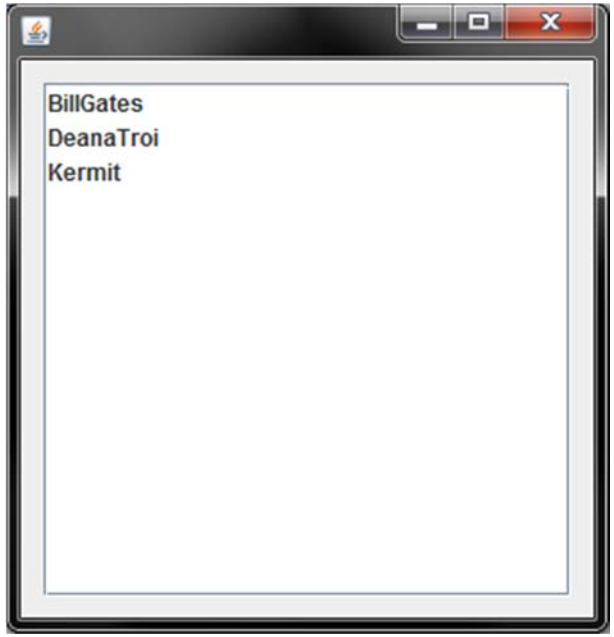
JList a modely

- **JList** je swingovský vizuálny komponent pre zobrazovanie položiek v podobe zoznamu



JList a modely

- **JList** zobrazuje dáta z modelu
- modelu zodpovedá interfejs **ListModel**



Model

- inštancia
- inštancia
- inštancia

JList a modely

- **JList** zobrazuje dáta z modelu
- modelu zodpovedá interfejs **ListModel**

```
interface ListModel {
  Object getElementAt(int index)
  int getSize();

  void addListDataListener(ListDataListener l);
  void removeListDataListener(ListDataListener l);
}
```

JList a modely

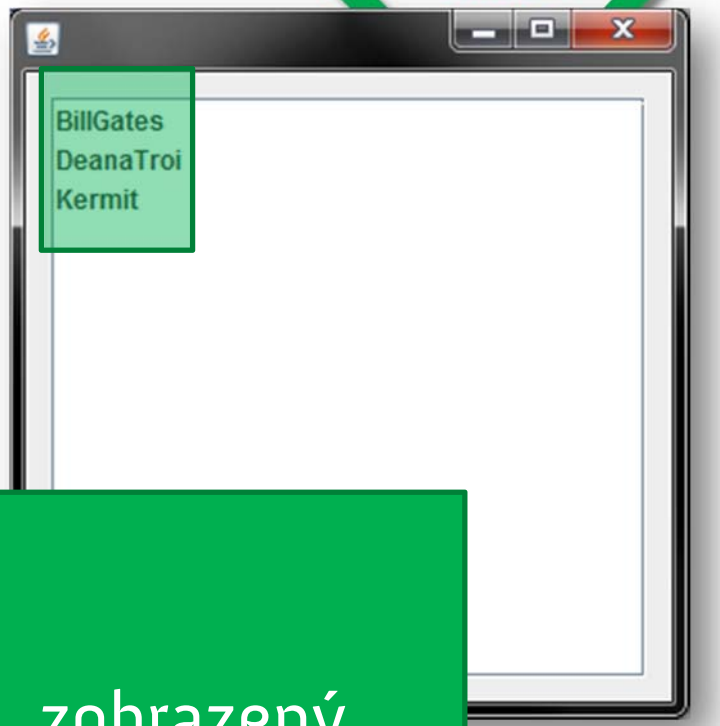
- **JList** zobrazuje dáta z modelu
- modelu zodpovedá interfejs **ListModel**
- buď implementujeme vlastnú triedu
- aleb štandardne využijeme niektorú preddefinovanú

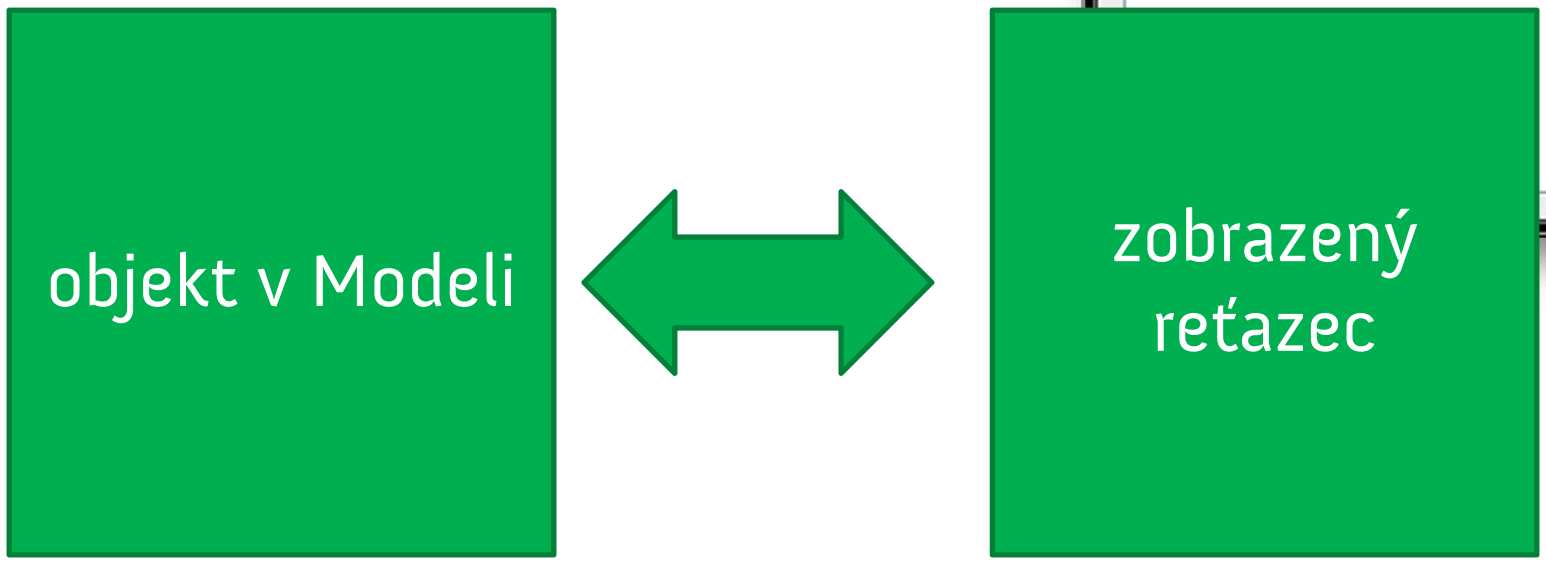
javax.swing.DefaultListModel

- **DefaultListModel** sa tvári približne ako **List**

JList a zobrazovanie prvkov z modelu

- model obsahuje **Object**-y
- JList však zobrazuje reťazce
- ako to?





JList a zobrazovanie prvkov z modelu

- ako sa dá previesť Object na reťazec?
- očividná možnosť: **.toString()**
- čo ak nie sme spokojní so štandardnou možnosťou?

javax.swing.ListCellRenderer

- Jlist má metódu **setCellRenderer()**

JList a zobrazovanie prvkov z modelu

- **ListCellRenderer** je **stratégia**, ktorá sa použije pri prevode objektu z modelu na reťazec

```
interface ListCellRenderer {
  Component getListCellRendererComponent(
      JList list,
      Object value,
      int index,
      boolean isSelected,
      boolean cellHasFocus);
}
```

Pre každú položku z modelu sa zavolá táto metóda.

Vrátiť sa má vizuálny komponent, ktorý zobrazí dáta.

Spoločný predok všetkých swingovských komponentov.

JList a zobrazovanie prvkov z modelu

- **ListCellRenderer** je **stratégia**, ktorá sa použije pri prevode objektu z modelu na reťazec
- opäť môžeme použiť vlastnú implementáciu

javax.swing.DefaultListCellRenderer

- na objekte zavolá **.toString()** a zobrazí ho ako **JLabel**
- zoznam **JList** totiž podporuje zobrazovanie zoznamov ľubovoľných komponentov

JList a zobrazovanie prvkov z modelu

- Čo ak chceme vlastnú stratégiu?
- Možnosť 1:
  - **prekryť metódu** na DefaultCellRendereri

```
public Component getListCellRendererComponent(
        JList list,
                                       Object value,
        int index,
        boolean isSelected,
        boolean cellHasFocus)
    {
        setComponentOrientation(list.getComponentOrientation());

        Color bg = null;
        Color fg = null;

        JList.DropLocation dropLocation = list.getDropLocation();
        if (dropLocation != null
                && !dropLocation.isInsert()
                && dropLocation.getIndex() == index) {

            bg = DefaultLookup.getColor(this, ui, "List.dropCellBackground");
            fg = DefaultLookup.getColor(this, ui, "List.dropCellForeground");

            isSelected = true;
        }

                                       if (isSelected) {
            setBackground(bg == null ? list.getSelectionBackground() : bg);
                                           setForeground(fg == null ? list.getSelectionForeground() : fg);
                                       }
                                       else {
                                           setBackground(list.getBackground());
                                           setForeground(list.getForeground());
                                       }

                                       if (value instanceof Icon) {
                                           setIcon((Icon)value);
                                           setText("");
                                       }
                                       else {
                                           setIcon(null);
                                           setText((value == null) ? "" : value.toString());
                                       }
                                       setFont(list.getFont());

        Border border = null;
        if (cellHasFocus) {
            if (isSelected) {
                border = DefaultLookup.getBorder(this, ui, "List.focusSelectedCellHighlightBorder");
            }
            if (border == null) {
                border = DefaultLookup.getBorder(this, ui, "List.focusCellHighlightBorder");
            }
        } else {
            border = getNoFocusBorder();
        }
                                       setBorder(border);

                                       return this;

    }
```

- **Problém**: kvôli jednému riadku musíme Ctrl-C, Ctrl-v celú metódu.
- Čo keď nemáme zdrojový kód?
- Čo keď v novej verzii opravia kód tejto metódy?

JList a zobrazovanie prvkov z modelu

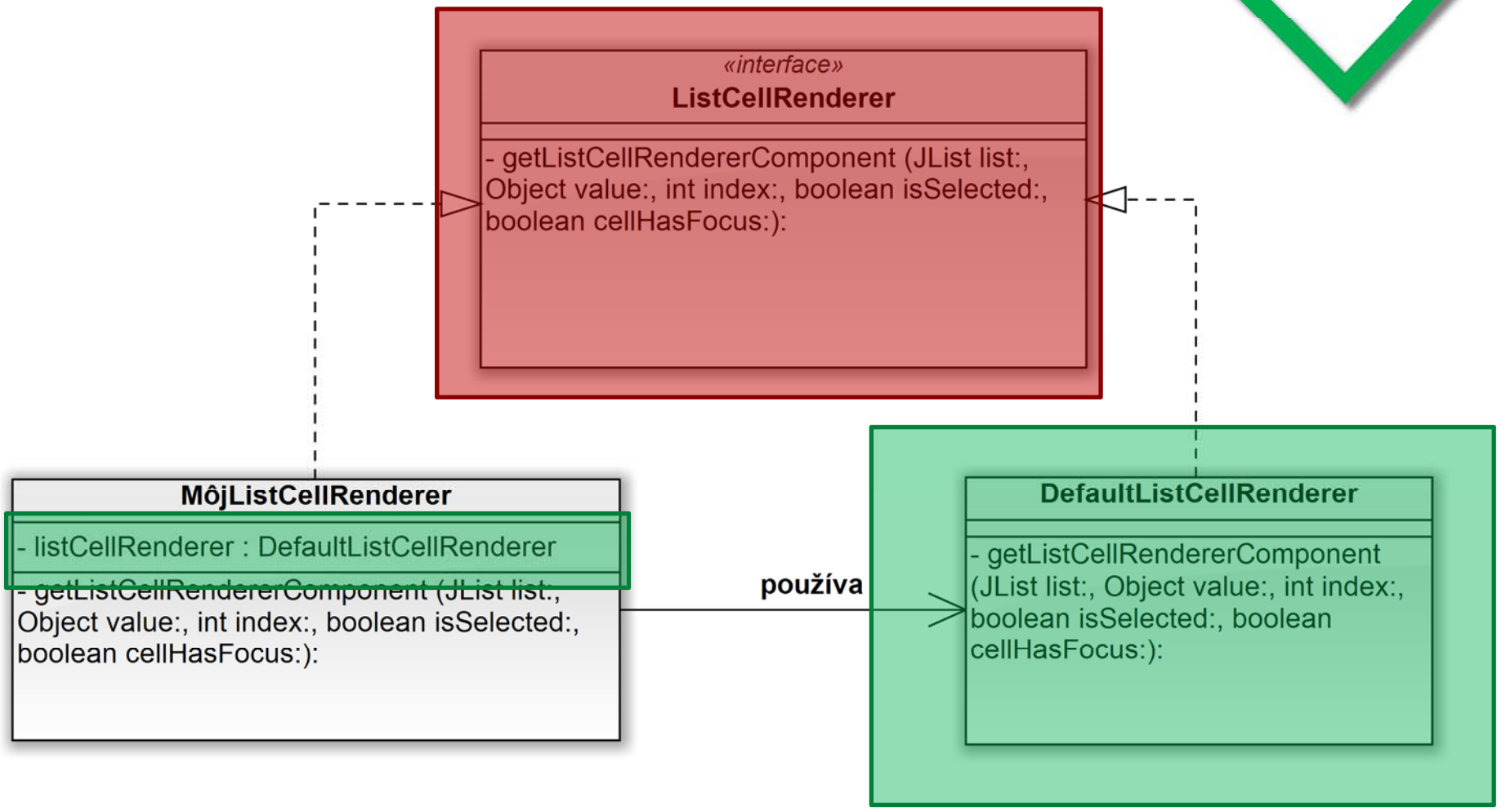
- Možnosť 2: použiť delegáciu
- **delegácia** = hybrid kompozície a dedičnosti
- vytvoríme triedu, ktorá implementuje interfejs ListCellRenderer
- trieda bude **delegovať** volanie na inštanciu **DefaultListCellRenderer**a
- môže využiť funkcionalitu hotovej triedy a v prípade potreby pozmeniť chovanie

JList a zobrazovanie prvkov z modelu


```
public class MôjListCellRenderer
   implements ListCellRenderer
{
  private DefaultListCellRenderer delegát;

   Component getListCellRendererComponent(...) {
     Kontakt kontakt = (Kontakt) value;
     return delegát
            .getListCellRendererComponent(..., kontakt, ...)
   }
}
```

- kombinácia **dedičnosti**
- a **kompozície**

Delegovanie



Delegácia

- delegácia kombinuje vlastnosti **dedičnosti** a kompozície
- trieda oddedí od inej triedy / implementuje interfejs
- schopnosti, ktoré chce **zmeniť**, zmení
- schopnosti, ktoré chce **znovupoužiť** z inej triedy, znovupoužije