Seminář Java

Zásady programování

Radek Kočí

Fakulta informačních technologií VUT

Duben 2008

Duplicitní objekty

Duplicitní objekty

- objekty mající stejný stav
- většinou neměnné
- vytváření objektů je zbytečné

Opakované použití shodného objektu

- může být rychlejší (optimální využití paměti) a přehlednější
- většinou použitelné pro neměnné objekty

Duplicitní objekty

```
String s = new String("retez");

String s = "retez";
```

```
Map m = new HashMap();

Set s1 = m.keySet();
Set s2 = m.keySet();
```

Dupl.java

Duplicitní objekty

Vytváření malých objektů

- malá funkcionalita kontruktorů
- rychlé (moderní implementace JVM)

Vytváření nových objektů

- může zlepšovat jednoduchost nebo sílu programu

Opakované použití objektu

- objekty jsou neměnné
- klesá výkon

Defenzivní kopie objektů

Defenzivní programování

- předpoklad, že klienti vaší třídy se pokusí zničit její invarianty
- neodkrývat interní prvky objektů
- před uložením provést defenzivní kopii
- před vrácením provést defenzivní kopii

Tovární metody místo konstruktorů

Získání instance třídy

- konstruktory
- statické tovární metody

Výhody továrních metod

- mají názvy
- nemusí vytvářet nový objekt při volání
- nemusí vracet instanci pouze volané třídy
- např. synchronizované kolekce, ...

Nevýhody

- těžko odlišitelné od jiných statických metod
- nutnost dodržování konvencí pojmenování

Tovární metody místo konstruktorů

```
public static final Boolean TRUE =
                            new Boolean(true);
public static final Boolean FALSE =
                           new Boolean(false);

public static Boolean valueOf(boolean b) {
    return (b ? TRUE : FALSE);
}
```

```
public static <T> Set<T> synchronizedSet(Set<T> s)
{
    return new SynchronizedSet<T>(s);
}
```

Tovární metody místo konstruktorů

```
public void m(Factory f) {
    Car car = f.getCar();
}

m(new FordFactory());
m(new VWFactory());

public class FordFactory extends Factory {
    public Car getCar() {
        return new FordCar();
    }
}
```

Překrývání společných metod

Třída **`Object`**

- nefinální metody **`equals`**, **`hashCode`**, **`toString`**, **`clone`**, **`finalize`**
- určené k překrytí v odvozených třídách

`equals`

- reflexivní:
  `x.equals(x) == true`
- symetrická:
  `x.equals(y) == true` $\Rightarrow$ **`y.equals(x) == true`**
- tranzitivní:
  `x.equals(y) == true` a **`y.equals(z) == true`** $\Rightarrow$
  `x.equals(z) == true`
- **`x.equals(null) == false`**

Symetrie metody `equals`

```
class CaseInsensitiveString {
   String s;
   public boolean equals(Object o) {
      if (o instanceof CaseInsensitiveString) ...
      if (o instanceof String)
         return s.equalsIgnoreCase((String) o);
      return false;
   }
}
```

```
CaseInsensitiveString cis =
          new CaseInsensitiveString("Pp");
String s = "pp";
cis.equals(s);
s.equals(cis);
```

Překrývání společných metod

- vždy když překryjete metodu **`equals`**, překryjte i metodu **`hashCode`**
- vždy překryjte metodu **`toString`**

Kompozice a dědičnost

Dědičnost

- nástroj znovupoužitelnosti kódu
- narušuje zapouzdření

Kompozice a dědičnost

```
class MyHashSet extends HashSet {
    private int addCount = 0;
    // konstruktory
    public boolean add(Object o) {
        addCount++;
        return super.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return super.addAll(c);
    }
    public int getCount() {
        return addCount;
    }
}
```

Kompozice a dědičnost

```
MyHashSet s = new MyHashSet();
s.addAll(Arrays.asList(
   new String[] {"jedna", "dva", "tri"} ));

s.getCount();       // => 6
```

- metoda **`addAll`** třídy **`HashSet`** používá metodu **`add`**
- implementační detail, který nemusí být dokumentovaný

Kompozice a dědičnost

```
MyHashSet s = new MyHashSet();
s.addAll(Arrays.asList(
   new String[] {"jedna", "dva", "tri"} ));

s.getCount();        // => 6
```

- metoda **`addAll`** třídy **`HashSet`** používá metodu **`add`**
- implementační detail, který nemusí být dokumentovaný

Kompozice a dědičnost

Dědičnost

- sémantika je založena na implementačních detailech rozšiřované třídy
- nová verze rozšiřované třídy může mít nové verze metod či nové metody
- problém překrývání metod
- náchylné na chyby

Kompozice a dědičnost

Kompozice

- nová třída se skládá (obaluje) původní třídu (resp. příslušné instance)
- metody jsou přesměrovány
- nová třída nebude záviset na implementačních detailech původní třídy
- problém SELF

Kompozice a dědičnost

```
class MyHashSet implements Set {
    private final Set s;
    private int addCount = 0;

    public MyHashSet(Set s) { this.s = s; }

    public boolean add(Object o) {
        addCount++;
        return s.add(o);
    }
    public boolean addAll(Collection c) {
        addCount += c.size();
        return s.addAll(c);
    }
```

Kompozice a dědičnost

```
    public int getCount() {
        return addCount;
    }

    // ostatni metody se musi presmerovat
    public void clear() { s.clear(); }
   ...
}
```

Rozhraní a abstraktní třídy

Definice typu, který umožňuje více implementací

- rozhraní
- abstraktní třída

Porovnání

- třídy lze snadno přizpůsobit tak, aby implementovaly rozhraní
- rozhraní může definovat smíšený typ
- rozhraní umožňují konstrukci nehierarchických typů
- rozhraní umožňují bezpečná vylepšení funkčnosti pomocí obalové třídy
- rozvíjet abstraktní třídu je mnohem jednodušší než rozvíjet rozhraní

Rozhraní a abstraktní třídy

Doporučení

- pro definici typů používejte (pokud to jde) vždy rozhraní
- změna implementace rozhraní pak znamená pouze změnu názvu konstruktoru (nebo tovární metody) bez nutnosti přepisovat další kód

Kontrola platnosti parametrů

- vždy kontrolujte platnost parametrů metod
- podmínky vždy dokumentujte

```
/**
 * Vraci BigInteger, jehoz hodnota je (this mod m).
 * @param m modulo, ktere musi byt kladne.
 * @return this mod m.
 * @throws ArithmeticException pokud m <= 0.
 */
public BigInteger mod(BigInteger m) {
   if (m.signum() <= 0)
      throw new ArithmeticException("m <= 0.");
   ...
}
```

Pole nulové délky

```
public String getContent() {
    if (content.size() == 0)
        return null;
    ...
}
```

- složitější implementace metody
- nutnost ošetřování po získání pole

```
String[] content = doc.getContent();
if (content != null) {
    ...
}
```

Pole nulové délky

- vracení `null` je efektivnější (nemusí se alokovat paměť na prázdné pole)?
- je možné vracet neměnný objekt nulové délky $\Rightarrow$ možnost sdílení

Pole nulové délky

```
private final static String[] NULL_ARRAY =
                                        new String[0];
public String getContent() {
    if (content.size() == 0)
        return NULL_ARRAY;
    ...
}
```

```
private final static String[] NULL_ARRAY =
                                        new String[0];
public String getContent() {
    return (String[]) content.toArray(NULL_ARRAY);
}
```

Přetěžování s rozvahou

- volba překryté metody (dědičnost) závisí na běhovém typu objektu (vybere se vždy ta nejspecifičtější varianta)
- volba přetížené metody se provádí při kompilaci

Přetěžování s rozvahou

```
public String classify(Set s)
    { return "Mnozina"; }
public String classify(List l)
    { return "Seznam"; }
public String classify(Collection c)
    { return "Neznama kolekce"; }

Collection[] test = new Collection[] {
    new HashSet(),
    new ArrayList(),
    new HashMap().values()
};
for (int i = 0; i < test.length; i++) {
    System.out.println(classify(test[i]));
}
```

Zastaralé odkazy na objekty

Paměťové úniky

- neúmyslné zachování již nepoužívaných objektů
- vyšší aktivita GC, vyšší spotřeba paměti
- nejsou zřejmé (špatně se odhalují)

Zdroj

- správa vlastní paměti
- cache

`Stack.java`

Zastaralé odkazy na objekty

Řešení

- nastavit `null`
- opakované použití proměnné
- definovat proměnnou v nejmenším možném oboru platnosti

Zvýšení výkonu (správa paměti)

Eliminace správy paměti

- zmenšit počet dočasných proměnných (v cyklech apod.)
- používat metody, které nevytvářejí dočasné objekty nebo nevracejí kopii objektu

```
String s = "55";
int i = new Integer(s).intValue();
int i = Integer.parseInt(s);
```

- místo řetězení + používat `StringBuffer`

`ObjCreation.java, Str.java`

Dokumentační komentáře