Seminář Java

XII

Radek Kočí

Fakulta informačních technologií VUT

25. dubna 2007

Obsah

- Zásady programování v jazyku Java
- Generics
- Reflektivní vlastnosti

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, Dupl2.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

`Period.java`

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

```
static public Block getInstance(String type) {
    Class cls;
    Block b = null;
    try {
        cls = Class.forName("editor.blocks."+type);
        b = (Block) cls.newInstance();
    }
    catch ...

    return b;
}
```

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

Deklarace generického rozhraní

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
```

```
public interface Iterator<E> {
    E next();
    boolean hasNext();
}
```

Generics – vyvolání

Vyvolání (invocation, parametrizovaný typ): **`List<Integer>`**

Generics v Javě

- jsou podobné šablonám (templates) v C++
- nejsou však stejné!
- nedochází ke generování nové verze třídy (rozhraní)

`E` ⇒ `Integer`

```
public interface List<Integer> {
    void add(Integer x);
    Iterator<Integer> iterator();
}
```

Generics – vyvolání

Generický kód

- kompiluje se pouze jednou
- parametrizovaný typ je podobný formálním parametrům metody
- metoda $\Rightarrow$ formální parametry hodnot
- generický kód $\Rightarrow$ formální parametry typů
- při vyvolání jsou formální parametry nahrazeny skutečnou hodnotou

---

```
public interface List<E> {
    void add(E x);
    Iterator<E> iterator();
}
```

Generics – dědičnost

```
/*1*/ List<String> ls = new ArrayList<String>();
/*2*/ List<Object> lo = ls;
/*3*/ lo.add(new Object());
/*4*/ String s = ls.get(0);
```

- `/*2*/` – `ls` a `lo` odkazují stejný objekt (kolekci)
- `/*3*/` – vložení instance třídy `Object` do kolekce
- `/*4*/` – výběr z kolekce, očekávaný typ `String`, skutečný `Object`!

- $\Rightarrow$ při `/*2*/` compile time error

Generics – dědičnost

```
/*1*/ List<String> ls = new ArrayList<String>();
/*2*/ List<Object> lo = ls;
/*3*/ lo.add(new Object());
/*4*/ String s = ls.get(0);
```

- **`/*2*/`** – **`ls`** a **`lo`** odkazují stejný objekt (kolekci)
- **`/*3*/`** – vložení instance třídy **`Object`** do kolekce
- **`/*4*/`** – výběr z kolekce, očekávaný typ **`String`**, skutečný **`Object`**!

- $\Rightarrow$ při **`/*2*/`** compile time error

Generics – dědičnost

```
/*1*/ List<String> ls = new ArrayList<String>();
/*2*/ List<Object> lo = ls;
/*3*/ lo.add(new Object());
/*4*/ String s = ls.get(0);
```

- **/*2*/** – **ls** a **lo** odkazují stejný objekt (kolekci)
- **/*3*/** – vložení instance třídy **Object** do kolekce
- **/*4*/** – výběr z kolekce, očekávaný typ **String**, skutečný **Object**!

- $\Rightarrow$ při **/*2*/** compile time error

Generics – dědičnost

- `Foo extends Bar`
- `G` je generická deklarace
- `G<Foo>, G<Bar>`
- `G<Foo>` *není podtřída* `G<Bar>`!

- `G` je jediná třída (rozhraní) s jedním parametrizovaným typem
- všechny instance parametrizované třídy `G` sdílejí stejnou třídu!

Generics – dědičnost

```
static void print(Collection<Number> cols) {
    for (Number o : cols)
        System.out.println(o);
}

Collection<Integer> ci = new ArrayList<Integer>();
print(ci); // compile-time error
```

Generics – ukázka

```
public class GenDemo<T> {
   private T value;
   private static T value2;
   public void put(T v) { value = v; }
   public T get() {
      return this.value;
   }
   public int add(int i) {
      return this.value.intValue() + 5;
   }
   public static void main(String[] argv) {
      GenDemo<Integer> gd = new GenDemo<Integer>();
      gd.put(new Integer(2));
      System.out.println(gd.get());
   }
}
```

Generics – ukázka

```
public class GenDemo<T> {
   private T value;
   private static T value2;
   public void put(T v) { value = v; }
   public T get() {
      return this.value;
   }
   public int add(int i) {
      return this.value.intValue() + 5;
   }
   public static void main(String[] argv) {
      GenDemo<Integer> gd = new GenDemo<Integer>();
      gd.put(new Integer(2));
      System.out.println(gd.get());
   }
}
```

Generics – ukázka

```
public class GenDemo<T extends Number> {

   private T value;
   public void put(T v) { value = v; }
   public T get() {
      return this.value;
   }
   public int add(int i) {
      return this.value.intValue() + 5;
   }
   public static void main(String[] argv) {
      GenDemo<Integer> gd = new GenDemo<Integer>();
      gd.put(new Integer(2));
      System.out.println(gd.get());
   }
}
```

Generics – wildcards

```
/*1*/
   void printCollection(Collection c) {
      Iterator i = c.iterator();
      for(int k = 0; k < c.size(); k++)
          System.out.println(i.next());
   }
```

```
/*2*/
   void printCollection(Collection<Object> c) {
      for(Object e :  c)
         System.out.println(e);
   }
```

Generics – wildcards

```
/*2*/
   void printCollection(Collection<Object> c) {
      for(Object e :  c)
         System.out.println(e);
   }
```

- **`/*2*/`** je méně univerzální než **`/*1*/`**
- **`/*2*/`**: **`Collection<Object>`** není nadtřídou ostatních kolekcí
- **`/*2*/`**: prvky kolekce mohou být pouze instance třídy **`Object`**

Generics – wildcards

Wildcard **?**

- reprezentuje neznámý typ

```
void printCollection(Collection<?> c) {
   for(Object e :  c)
      System.out.println(e);
}
...
Collection<String> cs = new
ArrayList<String>();
cs.add("Hi");
cs.printCollection(cs);
```

- lze číst, s prvky lze manipulovat jako s objekty (**Object**)

Generics – wildcards

Kolekce neznamych typu

- nelze predikovat typ prvku

```
Collection<String> cs = new ArrayList<String>();
cs.add("Hi");
cs.printCollection(cs);
...
void printCollection(Collection<?> c) {
   for(String e : c)  // => incompatible types
      System.out.println(e);
}
```

- nelze zapisovat

```
Collection<?> c = new ArrayList<String>();
c.add(new Object()); // compile time error
c.add(new String()); // compile time error
```

Generics – bounded wildcards

Příklad

- abstraktní třída **`Shape`**, metoda **`draw(Canvas c)`**
- rozšiřující třídy **`Rectangle`** a **`Circle`**
- třída **`Canvas`** s metodou **`drawAll`**

```
public void drawAll(List<Shape> shapes) {
    for(Shape s :  shapes)
        s.draw(this);
}
```

Problém

- prvky kolekce mohou být pouze instance třídy **`Shape`**
- nelze použít pouze **`<?>`**
- $\Rightarrow$ bounded wildcards

Generics – bounded wildcards

Bounded wildcards **<? extends T>**

- neznámá třída, která je potomkem třídy **T**
- může být i samotná třída **T**
- **T** je horní hranice

```
public void drawAll(List<? extends Shape> shapes)
{
   for(Shape s :  shapes)
      s.draw(this);
}
```

Generics – bounded wildcards

```
public void drawAll(List<? extends Shape> shapes)
{
    for(Shape s :  shapes)
        s.draw(this);
}
```

Problém

- nelze zapisovat do takto definované kolekce

```
public void addCircle(List<? extends Shape> shapes)
{
    shapes.add(new Circle()); // compile error
}
```

Generics – generické metody

Příklad (kopie pole do kolekce)

```
static void arrayToCol(Object[] a,
                       Collection<T> c) {
   for(Object o :  a)
      c.add(o); // compile time error
}
```

Problém

- neznámý typ prvků kolekce, nelze zajistit správnost
- $\Rightarrow$ parametrický typ **<T>**

```
static <T> void arrayToCol(T[] a,
                           Collection<T> c) {
   for(T o :  a)
      c.add(o);
}
```

Generics – inference typů

Volání předchozí metody

- **`void arrayToCol(T[] a, Collection<T> c)`**

Inference typů

- překladač odvozuje typy argumentů podle skutečně použitých

Generics – inference typů

```
void arrayToCol(T[] a, Collection<T> c) ...


Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
arrayToCol(oa, co);  // T is inferred to be Object

String[] sa = new String[100];
Collection<String> co = new ArrayList<String>();
arrayToCol(sa, cs);  // T is inferred to be String
arrayToCol(sa, co);  // T is inferred to be Object

Number[] na = new Number[100];
arrayToCol(na, cs);  // compile-time error
```

Generics – generické metody vs. wildcards

Příklad

- definice metod z rozhraní **Collection**

```
interface Collection<E> {
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c);
}
```

Možná modifikace

```
  public <T> boolean containsAll(Collection<T> c);
  public <T extends E>
        boolean addAll(Collection<T> c);
```

- na generický typ **T** není nic závislé
- je zbytečné ho uvádět
- použití wildcards je čistší a výstižnější

Generics – generické metody vs. wildcards

Příklad

- kombinace generické metody a wildcards
- metoda **`copy`** z třídy **`Collections`**

```
public static <T>
    void copy(List<T> dest, List<? extends T> src)
{
    ...
}
```

Generics – sdílení třídy

- generické třídy jsou sdílené pro všechny své objekty (invokace)
- nelze zajistit bezpečné přetypování na základě generických typů

Generics – sdílení třídy

```
List<String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
l1.getClass() == l2.getClass() // => true!

Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) ...
        // illegal generic type for instanceof

Collection<String> cstr = (Collection<String>) cs;
        // warning: [unchecked] unchecked cast
```

Generics – přetypování

- nelze zajistit bezpečné přetypování na základě generických typů

Generics – přetypování

```
<T> T badCast(T t, Object o) { return (T) o; }
// warning: [unchecked] unchecked cast

System.out.println(new GenericsDemo().
                   badCast("Hej", "Hou"));
// OK

System.out.println(new GenericsDemo().
                     badCast("Hej", 10));
// Exception in thread "main"
   java.lang.ClassCastException: java.lang.Integer
   at GenericsDemo.main(GenericsDemo.java:10)
```

Generics – bounded wildcards

Bounded wildcards **<? super T>**

```
interface Sink<T> { flush(T t); }
<T> T writeAll(Collection<? extends T> coll,
                                    Sink<T> snk)
{  T last;
   for(T t : coll) {
      last = t;
      snk.flush(last);
   }
   return last;
}

Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s);
```

Generics – bounded wildcards

- `T` $\Rightarrow$ `Object` $\Rightarrow$ špatný návratový typ

Generics – bounded wildcards

Bounded wildcards **<? super T>**

```
interface Sink<T> { flush(T t); }
<T> T writeAll(Collection<T> coll, Sink<? super T>
{
   T last;
   for(T t : coll) {
      last = t;
      snk.flush(last);
   }
   return last;
}

Sink<Object> s;
Collection<String> cs;
String str = writeAll(cs, s);
```

Reference

http://java.sun.com/j2se/1.5.0/docs/guide/language/index.html

http://java.sun.com/docs/books/tutorial/reflect/TOC.html

Reflektivita

Reflektivita

- zkoumání tříd a objektů
- speciální objekty reprezentující vlastnosti tříd a objektů
- každý element (třída, metoda, ...) má svou reprezentaci v podobě objektu
- `java.lang.Class<T>`
- `java.lang.reflect.Constructor<T>`
- `java.lang.reflect.Method`
- `java.lang.reflect.Field`

Reflektivita

Třída objektu

- každá zkompilovaná třída (bytecode) má proměnnou:
  `static public final class`
  př.:
  `java.lang.Class cls = Myclass.class;`
- metoda `Object.getClass()`
  ```
  String str = new String("Hi");
  Class cls = str.getClass();
  ```
- statická metoda `Class.forName()`
  ```
  Class cls =
  Class.forName("java.lang.Thread");
  ```

Reflektivita

Lze získat

- reprezentaci třídy, proměnné, metody, konstruktoru
- informace o názvu, modifikátoru třídy, proměnné, metody, konstruktoru
- informace o typu proměnné
- informace o navratovém typu metody
- informace o nadtřídě
- zda se jedná o třídu/rozhraní
- která rozhraní se implementují
- ...

Reflektivita

Je možné

- vytvářet nové instance tříd
- vyvolat metodu
- změnit obsah proměnné
- ...