Seminář Java

Generics

Radek Kočí

Fakulta informačních technologií VUT

Duben 2011

Generické programování

Parametrizované typy (generické programování)

- *templates* v C++ (viz Standard Template Library – STL)
- *generics* v Java 5
- definují parametrizované typy

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` $\Rightarrow$ **`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