Seminář Java
XI
2005/2006

Radek Kočí

Obsah

- Výčtový typ (enum)
- Proměnný počet argumentů (varargs)
- Statický import
- Reflektivní vlastnosti
- Generics

Enums

Reprezentace výčtového typu (Java < 5.0)

- int Enum pattern

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

- není typově bezpečný
- hodnoty nejsou informativní (vždy int; "jiné" názvy, stejná hodnota)

Výčtový typ (Java >= 5.0)

- typově bezpečný
- `enum Season { WINTER, SPRING, SUMMER, FALL }`
- plnohodnotná třída!
- rozšiřuje třídu `java.lang.Enum`

Enums

```
class EnumsDemo {
    public enum Season { WINTER, SPRING, SUMMER, FALL };
    private Season season;

    public static void main(String[] args) {
        new EnumsDemo();
    }

    public EnumsDemo() {
        season = Season.WINTER;
//        season = EnumsDemo.Season.WINTER;
        System.out.println(season);

        for (Season s : Season.values())
            System.out.println("Value of Season: " + s);
    }
}
```

Enums

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),
    VENUS   (4.869e+24, 6.0518e6),
    EARTH   (5.976e+24, 6.37814e6);

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
    public static final double G = 6.67300E-11;
    double surfaceGravity() {
        return G * mass / (radius * radius);
    }
    double surfaceWeight(double otherMass) {
        return otherMass * surfaceGravity();
    }
}
```

Enums

```
public static void main(String[] args) {
   double earthWeight = Double.parseDouble(args[0]);
   double mass = earthWeight/Planet.EARTH.surfaceGravity();

   for (Planet p : Planet.values())
      System.out.printf("Your weight on %s is %f%n", p,
                                        p.surfaceWeight(mass));
}
```

---

více na

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

Varargs

Varargs

- metoda s proměnným počtem parametrů
- pole objektů třídy `Object`

```
Object[] arguments = {
    new Integer(7),
    new Date(),
    "a disturbance in the Force"
};

String result = MessageFormat.format(
    "At {1,time} on {1,date}, there was {2} on planet "
     + "{0,number,integer}.", arguments);
```

Varargs (Java >= 5.0)

- vlastnost skrývající nutnost obalení polem
- poslední argument: `Object... name`

Varargs

```
class Args {

    public static void main(String[] args) {
        Args a = new Args();
        a.print("Hi ", "hou ", "ha");
    }

    public void print(String s, Object... args) {
        String str = "";
        System.out.println(args.length);
        for (Object o : args)
            str += (String) o;
        System.out.println(s + str);
    }
}
```

Static import

Přístup ke statickým členům

- `double r = Math.cos(Math.PI * theta);`
- Využití statického importu

```
import static java.lang.Math.PI;
//import static java.lang.Math.*;
...

double r = cos(PI * theta);
```

- používat velice opatrně! (kolize identifikátorů, těžko čitelný kód, ...)

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

Příklad

- získání jména třídy objektu

```
import java.lang.reflect.*;
import java.awt.*;

class SampleName {

   public static void main(String[] args) {
      Button b = new Button();
      printName(b);
   }

   static void printName(Object o) {
       Class c = o.getClass();
       String s = c.getName();
       System.out.println(s);
   }
}
```

Reflektivita

Příklad

- nadřazené třídy

```
static void printSuperclasses(Object o) {
   Class subclass = o.getClass();
   Class superclass = subclass.getSuperclass();

   while (superclass != null) {
      String className = superclass.getName();
      System.out.println(className);
      subclass = superclass;
      superclass = subclass.getSuperclass();
   }
}
```

Reflektivita

Příklad

- proměnné, modifikátory

```
static void printFieldNames(Object o) {
   Class c = o.getClass();
   Field[] fields = c.getDeclaredFields();

   for (int i = 0; i < fields.length; i++) {
      String fieldName = fields[i].getName();
      Class typeClass = fields[i].getType();
      String fieldType = typeClass.getName();

      int modif = fields[i].getModifiers();
      if (Modifier.isPublic(modif)) modstr += "public ";
      if (Modifier.isPrivate(modif)) modstr += "private ";
      if (Modifier.isStatic(modif)) modstr += "static ";
      ...
   }
}
```

Reflektivita

Příklad

- čtení/nastavení proměnné

```
Rectangle r = new Rectangle(100,20);
Integer widthParam = new Integer(300);
System.out.println("original: " + r.toString());

Field widthField;
Integer widthValue;
Class c = r.getClass();
try {
   widthField = c.getField("width");
   widthValue = (Integer) widthField.get(r);
   widthField.set(r, widthParam);
} catch (NoSuchFieldException e) { ... }
} catch (IllegalAccessException e) { ... }

System.out.println("modified: " + r.toString());
```

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
- ...

Reflektivita – vyvolání metody

```
class SampleMethods {

   public void first() {
      System.out.println("first method");
   }

   public void second(String s) {
      System.out.println(s);
   }

   public int third() {
      return 10;
   }

   ...
```

Reflektivita – vyvolání metody

```
...

public static void main(String[] args) {
   SampleMethods sm = new SampleMethods();
   Class c = sm.getClass();
   Method[] ma = c.getMethods();

   for (Method m : ma) {
      System.out.println("Method name: " + m.getName());
      if (m.getDeclaringClass() == SampleMethods.class)
      {
         invoke(m, sm);
      }
   }
}

...
```

Reflektivita – vyvolání metody

```
    private static void invoke(Method m, Object o) {
       try {
          if (m.getName().indexOf("first") != -1) {
             m.invoke(o);
          }
          if (m.getName().indexOf("second") != -1) {
             m.invoke(o, "hi");
          }
          if (m.getName().indexOf("third") != -1) {
             int i = (Integer) m.invoke(o);
             System.out.println("The result is " + i);
          }
       }
       catch (IllegalAccessException ex) { ... }
       catch (InvocationTargetException ex) { ... }
    }
}
```

Generics (opakování)

Problém při práci s kontejnery

- při vybírání elementu z kolekce musíme *přetypovat*
- není hlídáno kompilátorem
- pouze dynamická typová kontrola
- $\Rightarrow$ šance na vygenerování run-time výjimky

Generics (vlastnost Java 5.0)

- umožňuje komunikaci s kompilátorem
- statická kontrola typů při manipulaci s kontejnery
- bez přetypování!

Generics (opakování)

```
// Removes 4-letter words from c. Elements must be strings.
static void expurgate(Collection c) {
    for (Iterator i = c.iterator(); i.hasNext(); )
        if (((String) i.next()).length() == 4)
            i.remove();
}
```

```
// Removes the 4-letter words from c
static void expurgate(Collection<String> c) {
    for (Iterator<String> i = c.iterator(); i.hasNext(); )
        if (i.next().length() == 4)
            i.remove();
}
```

Generics – třída, rozhraní

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);
```

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`!

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!

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

Generics – ukázka

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

Generics – ukázka

```
public class GenericsDemo<T extends Number> {
   private T value;
   public void put(T value) {
      this.value = value;
   }
   public T get() {
      return this.value;
   }
   public int add(int i) {
      return this.value.intValue() + 5;
   }
   public static void main(String[] argv) {
      GenericsDemo<Integer> gd = new GenericsDemo<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);
   }
```

- `/*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 neznámých typů

- 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);
}
```

Problém

- nelze zapisovat do takto definované kolekce

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

Generics – generické metody

Příklad

- kopie pole do kolekce

```
static void arrayToCol(Object[] a, Collection<?> 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

```
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

```
interface Collection<E> {
   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

```
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

```
<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);
```

- `T` ⇒ `Object` ⇒ 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> 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);
```

- `T` ⇒ `String` ⇒ OK

Reference

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

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