Objektovo orientované programovanie

(generické typy)

9. prednáška (1. časť)

Vladislav Novák
FEI STU v Bratislave
11.11.2014

Obsah

Generické typy

príklad: Trieda slúžiaca pre odloženie referencie na inštanciu ľubovoľného typu

```
public class Schranka {
    private Object ulozenaInstancia;

    public void uloz(Object objekt) {
        ulozenaInstancia = objekt;
    }

    public Object daj() {
        return ulozenaInstancia;
    }
}
```

```
public static void main(String[] args) {
    Schranka schranka = new Schranka();
    schranka.uloz("retazec"); //OK pre lubovolny typ
    //.....

    String ret = (String) schranka.daj(); //kompilácia ok,
                      //ale za behu moze nastat chyba (tu nenastane)

    Integer cislo = (Integer) schranka.daj(); //kompilacia ok,
                      //ale za behu moze nastat chyba (tu nastane)
}
```

príklad: Trieda slúžiaca pre odloženie referencie na inštanciu s použitím generickej triedy

```
public class Schranka<T> {
    private T ulozenaInstancia;

    public void uloz(T objekt) {
        ulozenaInstancia = objekt;
    }

    public T daj() {
        return ulozenaInstancia;
    }
}
```

```
public static void main(String[] args){
 // Schranka<String> schranka = new Schranka<String>(); //od java 5
    Schranka<String> schranka = new Schranka<>(); //od java 7

    schranka.uloz("retazec"); //OK
    String ret = schranka.daj(); //OK (netreba pretypovat)

    schranka.uloz(10); //CHYBA pri preklade
    schranka.uloz(new Integer(10)); //CHYBA pri preklade
    Integer cislo1 = schranka.daj(); //CHYBA pri preklade
    Integer cislo2 = (Integer)schranka.daj();//CHYBA pri preklade
}
```

T je *premenná typu* (*type variable*), možno ju použiť na ľubovoľnom mieste v triede `Schranka`.
T je možné brať ako špeciálny typ premennej, ktorej hodnotou je typ.
Hodnotou premennej typu môže byť ľubovoľný referenčný typ.
Hodnotou premennej typu nemôže byť základný typ (využívajú sa obálkové triedy).

T je *parameter formálneho typu* triedy `Schranka`.

Rovnaký postup možno aplikovať aj na rozhrania
príklad:

```
public interface SchrankaRozhranie<T> {
    void uloz(T objekt);
    T daj();
}
```

```
public class SchrankaImplementacia<T>
                                    implements SchrankaRozhranie<T> {
    private T ulozenaInstancia;

    @Override
    public void uloz(T objekt) {
        ulozenaInstancia = objekt;
    }

    @Override
    public T daj() {
        return ulozenaInstancia;
    }
}
```

```
public static void main(String[] args){
    SchrankaRozhranie<String> schranka;
 // schranka = new SchrankaImplementacia<String>(); //od java 5
    schranka = new SchrankaImplementacia<>(); //od java7
    schranka.uloz("retazec");
    String ret = schranka.daj();
}
```

Volanie generického typu nahrádza symbol `T` konkrétnou triedou. Napr. volanie

```
Schranka<String> schranka;
```

deklaruje, že premenná `schranka` bude uchovávať referenciu na objekt typu `Schranka` s dosadeným *argumentom typu* `String`. Typ premennej `schranka` bude `Schranka<String>`.
Volanie generického typu sa obvykle označuje ako *parametrizovaný typ*.

Diamant
Vo verziách javy 5 a 6 (generické typy boli zavedené v jave 5) je pri vytváraní inštancie nutné uvádzať hodnotu parametra typu (pri volaní konštruktora).
Od javy verzie 7, v prípadoch kedy kompilátor dokáže odvodiť hodnotu parametra typu z kontextu, nemusíme uvádzať pri vytváraní inštancie hodnotu parametra typu. Stačí uviesť prázdnu množinu parametrov, t.j. prázdne zátvorky <> neformálne nazývané *diamant* (*diamond*). Toto sa označuje ako *automatické odvodenie typu* (*automatic type inference*).

Generická trieda obsahujúca viacero parametrov typu

príklad: generická trieda obsahujúca dva parametre typu

```
public class Dvojica<T,U> {
    private T prvy;
    private U druhy;

    public Dvojica() {
        this(null, null);
    }

    public Dvojica(T prvy, U druhy) {
        nastavImpl(prvy, druhy);
    }

    private void nastavImpl(T prvy, U druhy) {
        this.prvy = prvy;
        this.druhy = druhy;
    }

    public void nastav(T prvy, U druhy) {
        nastavImpl(prvy, druhy);
    }

    public T dajPrvy() {
        return prvy;
    }

    public U dajDruhy() {
        return druhy;
    }
}
```

```
public static void main(String[] args) {
    Dvojica<Integer,String> d1;
 // d1 = new Dvojica<Integer,String>(); //java 5, 6
    d1 = new Dvojica<>(); //od java7
    d1.nastav(10, "ahoj");
    System.out.println(d1.dajPrvy());
    System.out.println(d1.dajDruhy());

    Dvojica<Integer,String> d2;
 // d2 = new Dvojica<Integer,String>(20,"cau"); //java 5,6
    d2 = new Dvojica<>(20,"cau"); //od java 7
    System.out.println(d2.dajPrvy());
    System.out.println(d2.dajDruhy());
}
```

výstup:

```
10
ahoj
20
cau
```

Konvencie názvov parametrov typu

E – Element – prvok kolekcie
K – Key – kľúč – napr. v kolekcii typu Map
N – Number – číslo
T – Typ
V – Value - hodnota
S, U, V – druhý, tretí, štvrtý typ

Generické metódy a konštruktory

Parametre typu možno deklarovať aj v signatúrach metód a konštruktorov. Vytvoria sa tak *generické metódy* a *generické konštruktory*.
Pri volaní generickej metódy nemusíme explicitne udávať parametre typu. Prekladač ich odvodí podľa typu parametrov vložených pri volaní. Toto sa označuje ako *odvodenie typu* (*type inference*).
Generické metódy a generické konštruktory môžu byť v generických aj v nie generických triedach.

príklad:

```
public class GenerickeMetodyAKonstruktory {
    private static <T> void info(T prvok) {
        System.out.println("hodnota: " + prvok.toString());
        System.out.println("trieda: " + prvok.getClass().getName());
        System.out.println();
    }

    public static void main(String[] args) {
        info("retazec");
        info(new Integer(10));
        info(new Double(20));
    }
}
```

výstup:

```
hodnota: retazec
trieda: java.lang.String

hodnota: 10
trieda: java.lang.Integer

hodnota: 20.0
trieda: java.lang.Double
```

Podtypy

Vráťme sa k triede `Schranka`.
Trieda `Integer` je potriedou `Number`. Preto môžeme napísať:

```
Number number = new Integer(10); //OK
```

Trieda `Schranka<Integer>` ale nie je podtriedou `Schranka<Number>`, preto nemôžeme napísať:

```
//CHYBA pri preklade:
Schranka<Number> cisloA = new Schranka<Integer>();
```

Nasledujúci kód je ale v poriadku:

```
//toto je OK:

Schranka<Number> cisloB = new Schranka<Number>();

//implicitné pretypovanie Integer -> Number
cisloB.uloz(new Integer(100));

//volanie cisloB.daj() vráti hodnotu typu Number
//musíme explicitne pretypovať Number -> Integer
//explicitným pretypovaním vložíme kontrolu vykonanú za behu
Integer hodnota = (Integer) cisloB.daj();
```

Parametre viazaného typu

Java umožňuje obmedziť skupinu typov, ktoré možno predať parametru typu. Na obmedzenie typu zhora sa používa kľúčové slovo `extends`. Zápis

```
<T extends Number>
```

znamená, že parametrom typu `T` môže byť trieda `Number`, alebo jej podtrieda.
Za slovom `extends` môže nasledovať aj názov rozhrania.
Ak chceme uviesť za kľúčovým slovom `extends` triedu a aj rozhranie, alebo viacero rozhraní, potom treba použiť znak `&`.
príklad:

```
<T extends Number & MojeRozhranie>
```

príklad:

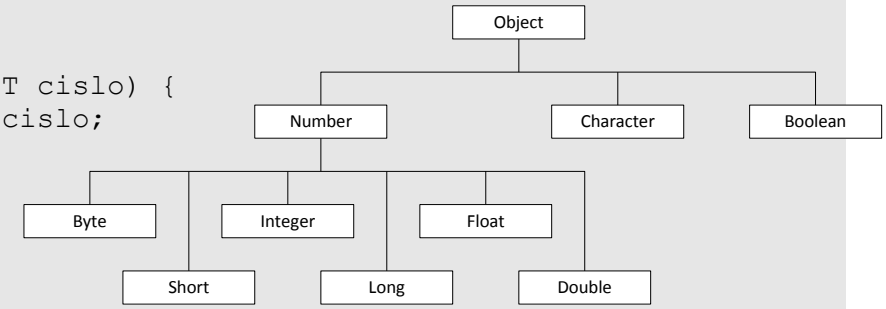
```
public class CiselnaSchranka <T extends Number> {
    private T cislo;

    public void uloz(T cislo) {
        this.cislo = cislo;
    }

    public T daj() {
        return cislo;
    }

    public int dajAkoInt() {
        return cislo.intValue(); //trieda Number obsahuje metódu
        // intValue(), preto inštancia typu T musí mať metódu intValue
    }

    public double dajAkoDouble() {
       return cislo.doubleValue(); //trieda Number obsahuje metódu
       //doubleValue(), preto inštancia typu T musí mať metódu doubleValue
    }
}
```




```
public static void main(String[] args) {
    CiselnaSchranka<Integer> cislo1= new CiselnaSchranka<Integer>();
    CiselnaSchranka<Double>  cislo2= new CiselnaSchranka<Double>();

    //CHYBA pri preklade:
  //CiselnaSchranka<String> retazec= new CiselnaSchranka<String>();
    cislo1.uloz(10);
    cislo2.uloz(new Double(20.1));

    System.out.println(cislo1.daj()); //10
    System.out.println(cislo2.daj()); //20.1

    System.out.println(cislo1.dajAkoInt()); //10
    System.out.println(cislo2.dajAkoInt()); //20
    System.out.println(cislo1.dajAkoDouble()); //10.0
    System.out.println(cislo2.dajAkoDouble()); //20.1
}
```

Zápis `<T>` znamená to isté ako `<T extends Object>`

Zástupné znaky

Vrátme sa teraz ku generickej triede:

```
public class Schranka<T> {
     .....
}
```

Ak poznáme dosadený argument typu, môžeme napísať deklarácie premenných (atribúty, lokálne premenné, vstupné parametre), alebo návratových typov napríklad:

```
Schranka<Integer> ciselnaSchranka;
```

alebo

```
public Schranka<Integer> metoda(Schranka<String> textovaSchranka) {
    // .....
}
```

V prípadoch keď nepoznáme aký bude dosadený argument typu, môžeme napísať:

```
Schranka<?> schranka;
```

alebo

```
public Schranka<?> metoda(Schranka<?> schranka) {
    // .....
}
```

Znak otáznik označuje neznámy typ, nazýva sa *zástupný znak* (*wildcard*).

Zápis `Schranka<?>` znamená, že dosadeným argumentom typu môže byť ľubovoľný typ

Zápis `Schranka<? extends Number>` znamená, že dosadeným argumentom typu môže byť typ `Number`, alebo ľubovoľný jeho podtyp (napr. `Integer`)

`<?>` má rovnaký význam ako `<? extends Object>`

príklad:

```
public static void main(String[] args) {
  //Schranka<Number> s1 = new Schranka<Integer>(); //CHYBA
    Schranka<?> s2 = new Schranka<Integer>(); //OK
    Schranka<?> s3 = new Schranka<String>();  //OK
    Schranka<?> s4 = new Schranka<Object>();  //OK
    Schranka<? extends Number> s5 = new Schranka<Number>(); //OK
    Schranka<? extends Number> s6 = new Schranka<Integer>();//OK
  //Schranka<? extends Number> s7 = new Schranka<String>(); //CHYBA
  //Schranka<? extends Number> s8 = new Schranka<Object>(); //CHYBA

  //s2.uloz(new Integer(10)); //CHYBA pri preklade
    Integer i2 = (Integer) s2.daj(); //musíme explicitne pretypovať

  //s5.uloz(new Integer(10)); //CHYBA pri preklade
    Integer i5 = (Integer) s5.daj(); //musíme explicitne pretypovať

  //s6.uloz(new Integer(10)); //CHYBA pri preklade
    Number n6 = s6.daj(); //OK
    Integer i6 = (Integer) s6.daj(); //musíme explicitne pretypovať
}
```

Pri volaní metódy `uloz()` prekladač nevie akého typu je argument metódy.

Zástupné znaky je ale možné využiť pri čítaní obsahu.

príklad: výpis implementovaných rozhraní

```
public static void main(String[] args) {
    Class<?> trieda = String.class;
    Class<?>[] rozhrania = trieda.getInterfaces();
    for(Class<?> rozhranie: rozhrania) {
        System.out.println(rozhranie.getName());
    }

    //kratsia verzia
    //for (Class<?> rozhranie : String.class.getInterfaces()) {
    //    System.out.println(rozhranie.getName());
    //}
}
```

výstup:

```
java.io.Serializable
java.lang.Comparable
java.lang.CharSequence
```

Obmedzenie typu zdola

Pre obmedzenie typu zdola sa používa kľúčové slovo `super`. Kód

```
<? super Number>
```

znamená, že typom môže byť trieda `Number`, alebo jej nadtriedy

príklady:

```
//Schranka<? super Number> s9 = new Schranka<Integer>(); //CHYBA

  Schranka<? super Number> s10 = new Schranka<Number>(); //OK
  s10.uloz(new Integer(10)); //OK
  Integer i10 = (Integer) s10.daj(); //musíme explicitne pretypovať

  Schranka<? super Number> s11 = new Schranka<Object>(); //OK
```

Prvotné typy (raw types)

Prvotný typ (*raw type*), je názov pre generickú triedu, alebo rozhranie zbavené argumentov typu. To znamená, že nie je možné zistiť, aký typ objektov bude inštancia generickej triedy pri behu používať. Používanie prvotných typov sa neodporúča, existuje len kvôli spätnej kompatibilite s kódmi písanými pred zavedením generických typov do jazyka java.

príklad :

```
Schranka schranka = new Schranka(); //neodporúča sa
```

`Schranka` je prvotný typ generického typu `Schranka<T>`
Použitie prvotného typu `Schranka` je to isté ako použitie `Schranka<Object>`

Trieda, alebo rozhranie ktoré nie je generické, nie je prvotný typ.

Mazanie typu

Pri vytváraní inštancie generického typu, používa prekladač metódu označovanú ako *mazanie typu* (*type erasure)*. Prekladač pritom odstráni všetky informácie súvisiace s parametrami typu a argumentmi typu v rámci triedy, alebo metódy. Aplikácie v jazyku Java, ktoré používajú genericitu, si tak môžu vďaka mazaniu typu udržať binárnu kompatibilitu s knižnicami a aplikáciami, ktoré vznikli skôr než sa genericita stala súčasťou jazyka java.

Generickú triedu `Schranka`:

```
public class Schranka<T> {
    private T ulozenaInstancia;

    public void uloz(T objekt) {
        ulozenaInstancia = objekt;
    }

    public T daj() {
        return ulozenaInstancia;
    }
}
```

kompilátor skompiluje nasledovne:

```
public class Schranka {
    private Object ulozenaInstancia;

    public void uloz(Object objekt) {
        ulozenaInstancia = objekt;
    }

    public Object daj() {
        return ulozenaInstancia;
    }
}
```

Generickú triedu

```
public class CiselnaSchranka <T extends Number> {
    private T cislo;

    public void uloz(T cislo) {
        this.cislo = cislo;
    }

    public T daj() {
        return cislo;
    }
    // ďalšie metódy
}
```

kompilátor skompiluje nasledovne

```
public class CiselnaSchranka {
    private Number cislo;

    public void uloz(Number cislo) {
        this.cislo = cislo;
    }

    public Number daj() {
        return cislo;
    }
    // ďalšie metódy
}
```

Obmedzenia pri používaní generických typov

Pri používaní generických typov existujú ďalšie obmedzenia.

Nie je možne deklarovať statický atribút parametra typu. Inštančné atribúty, lokálne premenné, parametre metóda a návratové typy môžu byť typu T.

```
public class GenerickaTrieda<T> {
    private static T atributTriedy; //CHYBA pri preklade
}
```

Nie je možné vytvoriť inštanciu parametra typu (ak nepoužijeme reflexiu). Tiež nie je možne vytvoriť pole prvkov parametra typu.

```
public <T> void metoda() {
    T premenna = new T(); //CHYBA pri preklade
    T[] pole1 = new T[10]; //CHYBA pri preklade
}
```

Nie je možné použiť operátor `instanceof` s parametrom typu

```
public <T> void metoda(Object objekt) {
  boolean test1 = objekt instanceof T; //CHYBA pri preklade
  boolean test2 = objekt instanceof Schranka<T>; //CHYBA pri prekl.
  boolean test3 = objekt instanceof Schranka<Integer>;//CHYBA prekl.
  boolean test4 = objekt instanceof Schranka<?>; //OK
}
```

Nie je možné vytvoriť pole prvkov parametrizovaných generických typov

```
Schranka<Integer>[] schranky = new Schranka<Integer>[10]; //CHYBA
                                                    // pri preklade
```

Nemožno preťažiť metódu tak, že po vymazaní typu, budú mať obidve metódy rovnakú signatúru (rovnaký prvotný typ).

```
//CHYBA pri preklade
public class Trieda {
    public void metoda(Schranka<Integer> ciselnaSchranka) { }
    public void metoda(Schranka<String>  textovaSchranka) { }
}
```

Nemožno definovať výnimku generického typu.

```
class Vynimka<T>  extends Throwable { } //CHYBA pri preklade
```

Nemožno zachytávať výnimku generického typu

```
public <T extends Exception> void metoda(T vynimka) {
    try {
        throw vynimka;
    }
    catch (T e) { //CHYBA pri preklade
    }
}
```

Avšak parametrický typ možno použiť v klauzule `throws`

```
public class Parser<T extends Exception> {
    public void parse(File file) throws T { //OK
    }
}
```

Príklad vytvorenia poľa prvkov parametrického typu

```
public class GenerickePole<T> {
    private T[] zoznam;

    public GenerickePole(int dlzkaPola) {
        zoznam = (T[]) new Object[dlzkaPola]; //VAROVANIE
                                  //na nekontrolovane pretypovanie
    }

    public void nastav(int index, T prvok) {
        zoznam[index] = prvok;
    }

    public T daj(int index) {
        return zoznam[index];
    }

    public int dlzka() {
        return zoznam.length;
    }
}
```

Pri pretypovaní `(T[])` vygeneruje kompilátor varovanie, pretože sa za behu pretypovanie nebude kontrolovať.

```
public static void main(String[] args) {
    GenerickePole<String> pole = new GenerickePole<String>(5);
    pole.nastav(0, "jeden");
    pole.nastav(1, "dva");
    pole.nastav(2, "tri");
    pole.nastav(3, "styri");
    pole.nastav(4, "pat");
    for (int i = 0; i < pole.dlzka(); i++) {
        System.out.println(pole.daj(i));
    }

    GenerickePole<Integer> pole2 = new GenerickePole<Integer>(10);
    for (int i = 0; i < pole2.dlzka(); i++) {
        pole2.nastav(i, i * 10);
    }
    for (int i = 0; i < pole2.dlzka(); i++) {
        System.out.println(pole2.daj(i));
    }
}
```

Výhody generických typov

Výhody oproti použitiu nie generických typov, kde by namiesto parametra typu bol použitý typ `Object`, alebo iný typ.

- pri používaní inštancii generických typov môže prekladač kontrolovať správnosť typov vstupných parametrov a návratových hodnôt.
- často nie je potrebné explicitne pretypovanie (nie je potrebné pretypovanie z `Object` (alebo iného typu) na používaný typ)