Generické datové typy a metody

Objektově orientované programování

Karel Šimerda

Fakulta elektrotechniky a informatiky

16. října 2019, Pardubice

Genericita

Co se naučíme

- Parametrizovat datové typy
- Jaká jsou specifika generických datových typů
- Bude ukázáno, jak se generické datové typy používají
- Jak s generickými datovými typy zachází překladač
- Jak předcházet konfliktům s představami překladače
- Parametrizovat metody
- Jak se pracuje se žolíky

Literatura

1. PECINOVSKÝ, Rudolf. Java 5.0: novinky jazyka a upgrade aplikací. Vyd. 1. Brno: CP Books, 2005. ISBN 80-251-0615-2. Strany 41-88
2. PECINOVSKÝ, Rudolf. Java 8: úvod do objektové architektury pro mírně pokročilé. Praha: Grada Publishing, 2014. Knihovna programátora (Grada). ISBN 978-80-247-4638-8. Strany 320-357
3. SCHILDT, Herbert. Java 8: výukový kurs. Brno: Computer Press, 2016. ISBN 978-80-251-4665-1. Strany 445-480

Motivace

- Již známe, že každá proměnná musí mít definován svůj datový typ
- Proto může už překladač zkontrolovat, že daná proměnná je správně používána
- U některých datových typů nevíme dopředu s jakým datovými typy bude pracovat
- Proto se začal takový datový typ definoval příliš obecně
- To zase mělo za následek, že nebylo možné mnoho věcí zkontrolovat
- Řešením bylo to, že se musela pro každý datový typ
  - zavádět nová třída se stejným chováním
  - přičemž to chování nebylo závislé na tom datovém typu
  - typickými představiteli jsou různé kolekce
- Javě trvalo přes 10 let než to vyřešila pomocí genericity
  - pro mnohé programátory je to obtížná látka
  - proto se ji budeme věnovat do větší hloubky

Pojmy parametrizovaných (generických) typů a metod

Generický datový typ je typem, který je parametrizován jiným datovým typem

Typový parametr je zástupce budoucí hodnoty datového typu v generickém datovém typu nebo metodě

Parametrizovaný typ vzniká odvozením od generického datového typu dosazením konkrétní hodnoty typového parametru

Generická metoda je metoda, kterou lze parametrizovat datovým typem

Parametrizovaná metoda je metoda třídy, která je samostatně parametrizována nějakým konkrétním datovým typem

Generické datové typy

- Jak odstranit problémy s množením tříd se stejným chováním pro různé datové typy?
  - Zavedení typových parametrů u datových typů
- Dosud jsme používali parametry v definicích metod
- Nyní lze používat parametry i u datových typů
- Typové parametry se deklaruji v ostrých (špičatých) závorkách například takto `<K,E>`
- Datové typy s typovými parametry označujeme jako generické datové typy
- V jazyce Java se generické datové typy používají od verze 5.0
- Slovo „generický“ v Javě znamená „obecný“
- V jazyku C++ se používají šablony (templates)
  - ty se opravdu generují jako nové třídy v době překladu

Ukázka torza třídy Seznam před verzí Javy 5.0

```
1  public class Seznam implements ISeznam {
2    private Object[] pole;   private int pocet;
3    public Seznam(int velikost) {
4      pole = new Object[velikost];
5    }
6    public void pridej(Object data) throws KolekceException {
7      Objects.requireNonNull(data, "Neni objekt");
8      if (pocet >= pole.length) {
9        throw new KolekceException("Prekroceni velikosti pole.");
10     }
11     pole[pocet++] = data;
12   }
13   public Object zpristupni(Object data) {
14     int index = hledej(data);
15     return (index == -1) ? null : pole[index];
16   }
17   public Object odeber(Object data) {
18     int index = hledej(data);
19     if (index == NO_INDEX) return null;
20     Object obj = pole[index];
21     posunPole(index);  pocet--;
22     return obj;
```

Použití třídy Seznam z předchozího slidu

- Následující kód obsahuje příliš široké možnosti přidávání objektů do seznamu

```
1  Seznam seznam = new Seznam(5);
2  seznam.pridej(new Stul());
3  seznam.pridej(new OsobniAuto);
4  seznam.pridej("Osobni auto");
5  seznam.pridej(5);
```

- Z uvedeného kódu vyplývá
  - do seznamu lze uložit odkaz na libovolný datový typ
  - protože se neprovádí žádná typová kontrola
  - překladač nemá žádnou informaci o typu
    - parametru metody `pridej(Object data )`
- Poznámka k řádku 5: Překladač automaticky zabalí hodnotu do objektu příslušné obalové třídy
  - tomu se říká „autoboxing“

Jak se pracuje typovými parametry

- Za názvem typu uvedeme v ostrých závorkách uvedeme identifikátor, který bude zastupovat budoucí datový typ

```
public interface GenerickyTyp<T>
```

- V místech použití generického typu se v ostrých závorkách uvede konkretní datový typ
- `GenerickyTyp<String> promenna = new GenerickyTyp<>()`
- Překladač si datový typ v prázdných závorkách domyslí podle typu proměnné
- Ostré prázdné závorky tvoří diamantový operátor
  - před verzí 7 jazyka Java, se musely závorky vždy vyplnit typem

Ukázka torza třídy Seznam od verze Javy 5.0

```
1   public class Seznam<E> implements ISeznam<E> {
2   private E[] pole;   private int pocet;
3   public Seznam(int velikost) {
4     pole = (E[])new Object[velikost];
5   }
6   public void pridej(E data) throws KolekceException {
7     Objects.requireNonNull(data, "Neni objekt");
8     if (pocet >= pole.length) {
9       throw new KolekceException("Prekroceni velikosti pole.");
10    }
11    pole[pocet++] = data;
12  }
13  public E zpristupni(E data) {
14    int index = hledej(data);
15    return (index == -1) ? null : pole[index];
16  }
17  public E odeber(E data) {
18    int index = hledej(data);
19    if (index == -1) return null;
20    E retData = pole[index];
21    posunPole(index);  pocet--;
22    return retData;
```

Použití třídy Seznam

- Následující kód obsahuje možnosti přidávání objektů do seznamu

```
1     Seznam<String> seznam = new Seznam<>(5);
2     //seznam.pridej(new Stul());
3     //seznam.pridej(new OsobniAuto);
4     seznam.pridej("Osobni auto");
5     //seznam.pridej(5);
```

- Z uvedeného kódu vyplývá
  - že do seznamu nelze uložit odkaz na libovolný datový typ
  - protože se provádí typová kontrola
  - v době překladu
    - tím, že za parametr metody `pridej(E data )` se dosadí datový typ `String`
    - a zkontroluje se s typem argumentu ve volání metody
    - pokud jsou typy kompatibilní, pokračuje se v překladu
  - po překladu jsou hodnoty parametrizovaných typů odstraněny

Konvence v označování typových parametrů

E	- element (používá se při kolekcích)
K	- key
N	- number
T	- typ
V	- value
S, U, V	atd. další typy

Parametrizované datové typy

- Vznikají odvozením od generického datového typu dosazením konkrétní hodnoty typového parametru
- Parametrizovaný datový typ: `GenerickyTyp<String>`

Definice generických typů

- Datový typ obsahuje v hlavičce seznam typových parametrů
- Typové parametry se potom používají v těle datového typu
- Lze je použít všude tam, kde lze v běžné třídě použít identifikátor datového typu
- Pomocí typových parametrů lze definovat:
  - typy atributů
  - typy parametrů metod
  - typy lokálních proměnných
  - typy návratových hodnot
  - typové parametry generických typů parametrů, návratových hodnot a lokálních proměnných instančních metod
- Typové parametry nelze použít, když překladač nemá potřebné informace
  - při volání konstruktoru
  - když není jasné, jaký konstruktor se má použít
  - jestli je vůbec nějaký konstruktor k dosažení

Rizika při vynechání nastavení typových parametrů

- Použití typových parametrů je nepovinné
  - Protože lidé (zejména studenti) jsou nepozorní a pohodlní
    - je použití konkrétních hodnot typu v při parametrizaci třídy nebo metody často vynecháváno
  - To potom způsobuje, že programátor je překladačem donucen doplňovat přetypování tam, kde to vlastně není potřeba
- Při důsledném používaní konkrétních hodnot typu v typových parametrech si překladač provede kontrolu na skutečný typ a přetypování potom už nevyžaduje

Další pojmy

- Typová kontrola
  - překladač ohlídá správný typ
  - snížení počtu přetypování
- Raw type
  - raw = základní, surový, čistý
  - typ po odstranění typových parametrů
  - raw typ ke generickému typu
    - `List<String>` je `List`
    - to znamená, že po překladu třída `List` pracuje datovým typem `Object`
- Type erasure
  - erasure = vymazání, očištění
  - proces odstranění typového parametru z generického typu
  - provádí jej překladač po kontrole typové kompatibility

Překlad a očišťování

- Hodnoty typových parametrů slouží pouze překladači
- V překladu o typových hodnotách už nic není
- Při překladu se pouze zkontroluje, zda si odpovídají datové typy
  - když si neodpovídají, překladač ohlásí chybu
  - když si odpovídají
    - kód se přeloží stejně, jako kdyby byl zapsán bez typových parametrů
    - tomu se říká očištění kódu
- Jinak řečeno
  - přeložený program je očištěn
  - po překladu nelze jednoduše zjistit jaká byla hodnota typového parametru
- To je důvod proč primitivní datové typy nemohou být hodnotou typového parametru - nelze je očistit na „raw“ typ
- Hodnoty primitivních datových typů se mohou použít v argumentech volání metod
  - překladač si provede konverzi hodnoty na příslušný obalový typ zadaný v typovém parametru

Typové parametry jen pro instanční členy

- Typové parametry z hlavičky generického typu nesmíme použít
  - ve statických atributech
  - ve statických metodách
- Důvodem je to, že hodnoty typových parametrů platí pro konkrétní instanci
- Různé instance mohou mít různé hodnoty typových parametrů
- Statické atributy a metody jsou společné pro všechny instance
- Statické členy musí mít vlastní typové parametry
  - u statických atributů to nemá smysl
  - naopak je tomu u statických metod, tam to někdy smysl má

Co nelze: Instanci typového parametru

- Nelze vytvořit instanci typového parametru

```
1    public class Zakazana<T> {
2        T t = new T();
3    }
```

- V tomto kódu nejsou dostatečné informace o typu, jehož instance se má vytvořit
- Instance lze ale získat:
  - prostřednictvím parametrů v metodách třídy
  - nebo výpočtem za běhu pomocí reflexe

```
1    public static class Container<E> {
2       private Class<E> clazz;
3       @SuppressWarnings("unchecked")
4       public Container(E instance) {
5         this.clazz = (Class<E>) instance.getClass();
6         E n = (E)clazz.getConstructors()[0].
              newInstance();
```

Řešení pomocí rozšíření jazyka Java verze 8.0

```
1  @FunctionalInterface
2  public interface Supplier<T> {
3      T get();
4  }
5  class SomeContainer<E> {
6      private Supplier<E> supplier;
7      public SomeContainer(Supplier<E> supplier) {
8          this.supplier = supplier;
9      }
10     public E createContents() {
11         return supplier.get();
12     }
13 }
14 SomeContainer<String> stringContainer =
15             new SomeContainer<>(String::new);
```

Co nelze: Pole instancí typového parametru

- Není možné vytvářet pole s prvky
  - typového parametru
  - parametrizovaného typu
- Pole si pamatuje typ svých prvků i po přetypování

```
1  public class Pole {
2    public static void procNelzePovolitPole() {
3      Seznam<String>[] sp =
4        // new Seznam<>[10]; //-- nelze
5        new Seznam[10];      //-- lze, ale !
6        Object obj = sp;
7        Object[] op = (Object[]) obj;
8        Seznam<Integer> si = new Seznam<>();
9        si.pridej(5);
10       op[0] = si;
11       String s = sp[0].dalsi(); // chyba az za behu
12     }
13   }
```

Pole instancí typového parametru, jiný příklad

```
1  class Gen <T extends Number>{
2    T obj;
3    T[] hodnoty;
4    public Gen(T o, T[] cisla) {
5      // hodnoty = new T[10];
6      hodnoty = cisla;
7      obj = o;
8    }
9  }
10 public class GenerickePole {
11   public static void main(String[] args) {
12     Integer n[] = {1, 2, 3, 4, 5};
13     Gen<Integer> objI = new Gen<Integer>(20, n);
14     //Gen<Integer> gener[] = new Gen<Integer>[10];
15     Gen<?> gener[] = new Gen<?>[10];
16     }
17   }
```

Co nelze: Parametrizovat výjimky

- Parametrizované typy nemohou být potomkem typu `Throwable`
  `public class MojeVyjimka<T> extends Exception`
- Při vyhození výjimky totiž není známo, kdo výjimku zachytí
- Důvodem je očištění od typových parametrů
- Proto parametrizace výjimky nemá smysl
- Typový parametr je možné použít k vyhození výjimky,
  ale nelze ho použít k zachycení výjimky v klauzuli **`catch`**

Nejednoznačnosti a kolize

- Očištěním se ztratí spousta informací
- Proto může docházet
  - k nejednoznačnostem
  - a ke kolizím

Kolize: Proč překladač odmítne tuto třídu přeložit?

```
1  public class Prepravka<T1, T2> {
2
3    T1 prvni;
4    T2 druhy;
5
6    public T1 get1() { return prvni; }
7
8    public T2 get2() { return druhy; }
9
10   public void set(T1 o){ prvni = o; }
11
12   public void set(T2 o){ druhy = o; }
13
14 }
```

Kolize: Proč překladač odmítne i tuto třídu přeložit?

```
1  public class Prepravka<T> {
2
3      T prvni;
4      T druhy;
5
6      public boolean equals(T t){
7          return (prvni.equals(t) && druhy.equals(t));
8      }
9  }
```

Generické metody

- Jako generické můžeme definovat nejenom datové typy
  - ale i jednotlivé metody
- Typové parametry se uvádějí před typem návratové hodnoty
- Při volání takovéto metody skutečnou hodnotu
  - typového parametru si překladač často domyslí;
  - pokud ne, musíme ji uvést
- Skutečnou hodnotu typového parametru píšeme za kvalifikaci
  - potřebujeme-li uvést typový parametr, musíme metodu kvalifikovat,
  - i když by to jinak nebylo potřeba

Ukázka použití generické metody

```
1  public class Metody {
2
3    public static <T> List<T> dvojice( T prvek ) {
4      List<T> list = new ArrayList<>();
5      list.add( prvek );  list.add( prvek );
6      return list;
7    }
8
9    public void volani() {
10     Metody m = new Metody();
11     List<String> ls = dvojice("Text");
12     List<Integer> lo;
13   // lo = dvojice("Text");       // Nesouhlasi typy
14   // lo = <Object>dvojice(1));  // Nekvalifikovane
15     lo = Metody.dvojice(2);     // Kvalifikace tridou
16     lo = m.dvojice(3);          // Kvalifikace instanci
17     lo = dvojice((Integer(4)); // Reseni pretypovanim
```

Meze typových parametrů

- Specifikují úžeji třídu nebo rozsah tříd typových argumentů
- Jak se zapisuje:
  - `T extends HorniMez`
  - T může být HorniMez nebo její potomek
- Zúžení typu parametrů může být
  - absolutní:
    ```
    class Trida <T extends Number> ...
    ```
  - vzájemné:
    ```
    class Trida <S, T extends S> ...
    ```
- Není-li mez uvedena, je mezí `Object`.
- Kromě horního limitu lze specifikovat i dolní limit
  - `class Trida <T super Integer> ...`
  - potom budou přípustné pouze argumenty tříd, které jsou nadtřídami příslušné podtřídy

Ukázka dědičnosti parametrických typů

```
1  public class FrontaPI<E> extends FrontaP<E>
       implements Iterable<E> {
2    public Iterator<E> iterator() {
3      return new Iterator<E>() {
4        int poradi = 0;
5        public boolean hasNext() {
6          return (poradi < prvky.size());
7        }
8        public E next() {
9          return prvky.get( poradi++ );
10       }
11       public void remove() {
12         prvky.remove( 0 );
13       }
14     };
```

Žolíky

- Slouží k řešení problémů způsobených omezeními dědičnosti parametrizovaných typů
- Může-li na daném místě být objekt libovolného typu, použijeme žolík
  - `Class<?> trida = parametr.getClass();`
- Může-li být na daném místě objekt libovolného typu vyhovujícího zadanému omezení, použijeme žolík s příslušným omezením
  - Pro potomka typu X: `<? extends X>`
  - Pro rodiče typu Y: `<? super Y>`

  Omezení lze kombinovat:
  - `<? extends Comparable<? super T>>`

Ukázka

- Tisk kolekce ve starších verzí Javy

```
1     public static tiskniSeznam (Seznam s){
2         for(Iterator it = s.iterator; s.hasNext; ) {
3             System.out.println(it.next());
4         }
5     }
```

- To samé, ale ve verzi 5 a vyšší - tiskne jen kolekce s prvky Object

```
1     public static tiskniSeznam (Seznam<Object> s){
2         for(Iterator it = s.iterator; s.hasNext; ) {
3             System.out.println(it.next());
4         }
5     }
```

- To samé, ale se žolíkem - bude tisknout prvky podle bez ohledu na jejich typ

```
1     public static tiskniSeznam (Seznam<?> s){
2         for(Iterator it = s.iterator; s.hasNext; ) {
3             System.out.println(it.next());
4         }
5     }
```

Omezení hodnot žolíků

- Žolík jako potomek zadaného typu

```
1     List<? extends INabytek> seznam;
2     seznam = new ArrayList<Stul>();
3     seznam = new LinkedList<Skrin>();
```

- Žolík jako předek zadaného typu

```
1     public class Trida<T extends
2                             Comparable< ? super T>>
```

Děkuji za pozornost!