
Kapitola 1. Přednáška 9 - dokončení dynamických datových struktur (Mapy). Generické typy.

Obsah

Generické typy	1
Generické datové typy (generics)	1
Základní syntaxe	2
Jednoduché využití v metodách	2
První příklad použití	3
Cyklus foreach	4
Žolíci (wildcards)	5
Generické metody	8
Generics metody vs. wildcards	8
Pole	9
Vícenásobná vazba generics	9
Závěr	10

Generické typy

Generické datové typy (generics)

Pokud si vezmeme anglicko-český slovník, zjistíme, že v překladu to znamená něco obecně použitelného, tedy mohli bychom použít termínu *zobecnění*. A přesně tím generics jsou -- zobecněním.

Jak již víme, struktura tříd v Javě má společného předka, třídu `Object`. Tato skutečnost zjednodušuje implementaci nejednoho programu -- potřebujeme-li pracovat s nějakými objekty, o kterých tak úplně nevíme, co jsou zač, můžeme využít společného předka a pracovat s ním. Každý objekt v programu je totiž i instancí třídy `Object`. To v praxi umožňuje například snadnou implementaci spojových seznamů, hashovacích tabulek, ale například i využití reflexe.

Jakkoliv je společný předek jistě výhodou, přináší i některé obtíže. Uvažujme opět (spojový) seznam. Pracujeme-li v programu s nějakým, víme jenom, že je to seznam objektů a nic více. Samozřejmě si můžeme bokem pamatovat, že tam ukládáme jenom řetězce, ale to nic nemění na tom, že runtime systému je srdečně jedno, jaké objekty do seznamu vkládáme. Stejně tak, chceme-li z tohoto seznamu číst, získáváme zase jenom obecné objekty, které musíme explicitně přetypovat na řetězec (tj. třídu `String`). A opět -- ačkoliv jako programátoři víme, co v seznamu *má být*, obecně si nemůžeme být jisti, zda to tam skutečně *je*. Což vede buď k tomu, že se budeme ptát, zda získávaný objekt je skutečně řetězcem a až poté jej přetypujeme nebo budeme „riskovat“ runtime výjimku `ClassCastException`. Jak vidíme,

nevýhody jsou a to docela významné.

Řešením v této situaci je příchod Javy verze 1.5, která mimo jiné přináší i zmiňovaná *generics*. Opět využijeme našeho spojového seznamu. Vraťme se zpět k jeho definici. Co od něj vlastně požadujeme?

- Aby byl seznamem čehokoliv a tak byl universální (což je stav popsán výše)?
- Nebo aby to byl seznam nějakého obecného, předem nedefinovaného typu a umožnil tento typ při vytvoření instance určit?

Jelikož předchozí otázky jsou řečnické, odpověď následuje ihned. Samozřejmě, že druhá uvedená možnost je lepší (už proto, že má veškerou sílu první možnosti, stačí nadefinovat, aby tím obecným typem byl `Object`). Ve zbytku textu si tedy ukážeme, jak toho generics docílují a jak je používat.

Základní syntaxe

V úvodu jsme se lehce zmínili o spojovém seznamu. Nyní si budeme ukazovat příklady na obecném seznamu (rozhraní `java.util.List`).

Takhle vypadá deklarace `List` bez použití generics:

```
public interface List {  
    ...  
}
```

A takhle s nimi:

```
public interface List <E> {  
    ...  
}
```

Jak vidíme, úvod je velmi jednoduchý. Do špičatých závorek pouze umístíme symbol, kterým říkáme, že seznam bude obsahovat prvky *E* (předem neznámého) typu.

Zde uděláme malou odbočku -- je doporučováno používat velké, jednopísmenné deklarace generics. To-to písmeno by zároveň mělo vystihovat použití resp. význam takového zobecnění. Tedy *T* je typ, *E* je prvek (element) a tak podobně

Jednoduché využití v metodách

Pouhá deklarace u jména třídy resp. rozhraní samozřejmě nemůže stačit. Zjednodušeně řečeno, zdrojový kód využívající generics musí typ *E* použít všude tam, kde by dříve použil obecný `Object`. To jest například místo

```
Object get(int index);
```

se použije

```
E get(int index);
```

Co jsme nyní udělali? Touto definicí jsme řekli, že metoda `get` vrací pouze objekty, které jsou typu *E* na místo libovolného objektu, což je přesně to, co od generics vyžadujeme. Všimněte si, že nyní už s *E* pracujeme jako s jakoukoliv jinou třídou nebo rozhraním.

Totožně postupujeme i u metod, které do seznamu prvky typu *E* přidávají. Viz

```
boolean add(E o);
```

Dovolím si další malou poznámku na okraj -- výše zmíněné metody by samozřejmě mohly pracovat s typem `Object`. Překladač by proti tomu nic nenamítal, nicméně očekávaná funkcionality by byla pryč.

První příklad použití

Nyní tedy máme seznam, který při použití bude obsahovat nějaké prvky typu *E*. Nyní chceme takový seznam použít někde v našem kódu a užívat si výhod generics. Vytvoříme jej následovně:

```
List<String> = new ArrayList<String>();
```

Použití je opět jednoduché a velmi intuitivní. Nyní následují dva příklady demonstrující výhody generics (první je napsán „postaru“).

```
Object number = new Integer(2);
List numbers = new ArrayList();
numbers.add(new Integer(1));
numbers.add(number);
Number n = (Number)numbers.get(0);
Number o = (Number)numbers.get(1);
```

```
Object number = 2;
List<Number> numbers = new ArrayList<Number>();
numbers.add(1);
numbers.add((Number)number);
Number n = numbers.get(0);
Number o = numbers.get(1);
```

Jak vidíme v horním příkladu, do seznamu lze vložit libovolný objekt (byť zde jsme měli „šťěstí“ a bylo to číslo) a při získávání objektů se spoléháme na to, že se jedná o číslo. Níže naopak nelze obecný objekt vložit, je nutné jej explicitně přetypovat na číslo, teprve poté překladač kód zkompiluje. Podotkněme, že pokud bychom na `Number` přetypovali například `String`, program se také přeloží, ale v okamžiku zavolání takového příkazu se logicky vyvolá výjimka `ClassCastException`. Získání čísel ze seznamu je ovšem přímočaré -- stačí pouze zavolat metodu `get`, která má správný návratový typ.

Povšimněte si rovněž použití další vlastnosti nové Javy, tzv. *autoboxingu*, kdy primitivní typ je automaticky převeden na odpovídající objekt (a vice versa).

Cyklus foreach

Přestože konstrukce cyklu **for** patří svou povahou jinam, zmiňujeme se o nich zde, u dynamických struktur - kontejnerů, neboť se převážně používá k iterování (procházení) prvků seznamů, množin a dalších struktur. Obecný tvar cyklu "foreach" je syntaktickou variantou běžného "for":

```
for (TypRidiciPromenne ridici_promenna : dyn_struktura) {
    // co se dela pro kazdou hodnotu ze struktury...
}
```

V následujícím příkladu jsou v jednotlivých průchodech cyklem **for** postupně ze seznamu vybírány a do řídicí proměnné *e* přiřazovány všechny jeho prvky (objekty).

```
for (Object e : seznam) {  
    System.out.println(e);  
}
```

Žolíci (wildcards)

V předchozích částech jsme se seznámili s hlavní myšlenkou generics a její realizací, a sice nahrazení konkrétního nadtypu (většinou `Object`) typem obecným. Nicméně tohle samo o sobě je velmi omezující a nedostačující. Nyní se tedy ponoříme hlouběji do tajů generics.

Představme si následující situaci. V programu chceme mít seznam, kde budou jako prvky různé jiné seznamy. První nápad, jak jej nadeklarovat může být třeba tento:

```
List<List<Object>> seznamSeznamu;
```

Na první pohled se to zdá být bez chyby. Máme seznam, kam budeme vkládat jiné seznamy a jelikož každý seznam musí obsahovat instance třídy `Object`, můžeme tam vložit libovolný seznam, tedy třeba i náš `List<Number>`. Nicméně tato úvaha je chybná. Uvažujme následující kód:

```
List<Number> cisla = new ArrayList<Number>();  
List<Object> obecny = cisla;  
obecny.add("Ja nejsem cislo");
```

Jak vidíme, „něco je špatně.“ To, že se pokoušíme přiřadit do seznamu objektů `obecny` řetězec "Ja nejsem cislo" je přece naprosto v pořádku, do seznamu objektů můžeme skutečně vložit cokoliv. V tom případě ale musí být špatně přiřazení na druhém řádku. To znamená, že seznam čísel *není* seznamem objektů! Zde je vidět rozdíl oproti „klasickému“ uvažování v mezích dědičnosti. Přečtěte si pozorně následující větu a pokuste se pochopit její význam.

Do seznamu, který obsahuje nejvýše čísla lze vkládat pouze objekty, které jsou alespoň čísla.

Z toho vyplývá, že je nelegální přiřazovat objekt „seznam čísel“ do objektu „seznam objektů.“ Tedy,

vrátíme-li se k našemu příkladu se seznamem seznamů, vidíme, proč byla naše úvaha chybná. Do námi definovaného seznamu totiž lze ukládat pouze seznamy objektů a ne libovolné seznamy. Jak tedy docílíme kýženého jevu? K tomuto účelu nám generics poskytují nástroj zvaný *žolík*, anglicky *wildcard*, který se zapisuje jako ?. Vraťme se nyní k předchozímu příkladu:

```
List<Number> cisla = new ArrayList<Number>();
List<?> obecny = cisla;           // tohle je OK
obecny.add("Ja nejsem cislo");    // tohle nelze prelozit
```

Jak je již v komentáři kódu naznačeno, poslední řádek neprojde překladačem. Proč? Protože pomocí `List<?>` říkáme, že `obecny` je seznamem *neznámých* prvků. A jelikož nevíme, jaké prvky v seznamu jsou, nemůžeme do něj ani *žádné* prvky přidávat. Jedinou výjimkou je „žádný“ prvek, totiž `null`, který lze přidat kamkoliv. Mírně filosoficky řečeno, *null není ničím a tak je zároveň vším*.

Naopak, ze seznamu neznámých objektů můžeme samozřejmě prvky číst, neboť každý prvek je určitě alespoň instancí třídy `Object`. Ukážeme si praktické použití žolíku.

```
public static void tiskniSeznam(List<?> seznam) {
    for (Object e : seznam) {
        System.out.println(e);
    }
}
```

Nyní si představme, že chceme metodu, která udělá z nějakého seznamu čísel jeho sumu. Uvažujme tedy následující (a pomeňme možné přetečení nebo podtečení rozsahu `double`):

```
public static double suma(List<Number> cisla) {
    double result = 0;
    for (Number e : cisla) {
        result += e.doubleValue()
    }
    return result;
}
```

Opět, metoda se jeví jako bezproblémová. Nic ale není tak jednoduché, jak by se mohlo zdát. Nyní zku-

síme uvažovat bez příkladu. Představme si, že máme seznam celých čísel, u kterého chceme provést sumu. Jistě není sporu o tom, že celá čísla jsou zároveň obecná čísla a přesto seznam `List<Integer>` nelze použít jako parametr výše deklarované metody z naprosto stejného důvodu, kvůli kterému nešlo říci, že seznam objektů je seznam čísel.

Samozřejmě je tu opět řešení. Zkusme nejdříve uvažovat selským rozumem. Výše jsme říkali, že místo *seznamu objektů* chceme *seznam neznámých prvků*. Nyní jsme v podobné situaci, pouze se nacházíme na jiném místě v hierarchii tříd. Zkusme tedy obdobnou úvahu použít i zde. Nechceme *seznam čísel* nýbrž *seznam neznámých prvků, které jsou nejvýše čísla*. Nyní je již pouze třeba ozřejmit syntaxi takové „úvahy“.

```
public static double suma(List<? extends Number> cisla) {  
    ...  
}
```

Toto použití žolíku má uplatnění i v samotném rozhraní `List<E>` a sice v metodě „přidej vše“. Zamyslete se nad tím, proč tomu tak je.

```
boolean addAll(Collection<? extends E> c);
```

Uvědomte si prosím následující -- prostý žolík je vlastně „zkratka“ pro „neznámý prvek rozšiřující `Object`“.

Ač by se tak mohlo zdát, možnosti *wildcards* jsme ještě nevyčerpali. Představme si situaci, kdy potřebujeme, aby možnou hodnotou byla instance třídy, která je v hierarchii mezi třídou specifikovanou naším obecným prvkem *E* a třídou `Object`. Pokud přemýšlíte, k čemu je něco takového dobré, představte si, že máte množinu celých čísel, které chcete setřídít. Jak lze taková čísla třídít? Například obecně podle hodnoty metody `hashCode()`, tedy na úrovni třídy `Object`. Nebo jako obecné číslo, tj. na úrovni třídy `Number`. A konečně i jako celé číslo na úrovni třídy `Integer`. Skutečně, níže již jít nemůžeme, protože libovolné zjemnění této třídy například na celá kladná čísla by nemohlo třídít obecná celá čísla.

Následující příklad demonstruje syntaxi a použití popsané konstrukce

```
public TreeMap(Comparator<? super K> c);
```

Jedná se o konstruktor stromové mapy, tj. mapy klíč/hodnota, která je navíc setříděna podle klíče. Nyní

opět trochu odbočíme a podíváme se, jak vypadá deklarace obecného rozhraní seřazené mapy.

```
public interface SortedMap<K,V> extends Map<K,V> {...
```

Máme zde nový prvek -- je-li třeba použít více nezávislých obecných typů, zapíšeme je opět do „zobáčků“ jako seznam hodnot oddělených čárkou. Povšimněte si opět mnemotechniky -- *K* je *key* (klíč), *V* je *value* (hodnota). Je-li to třeba, je možné použít i žolíků. Viz následující příklad konstruktorů naší staré známé stromové mapy.

```
public TreeMap(Map<? extends K, ? extends V> m);  
public TreeMap(SortedMap<K, ? extends V> m);
```

Generické metody

Tato část bude relativně krátká a stručná, poněvadž pro používání *generics* a žolíků platí stále stejná pravidla. Generickou metodou rozumíme takovou, která je parametrizována alespoň jedním obecným typem, který nějakým způsobem „váže“ typy proměnných a/nebo návratové hodnoty metody.

Představme si například, že chceme statickou metodu, která přenesení prvky z pole nějakého typu přidá hodnoty do seznamu s prvky téhož typu.

```
static <T> void arrayToList(T[] array, List<T> list) {  
    for (T o : array) {  
        list.add(o);  
    }  
}
```

Zde narážíme na malou záludnost. Ve skutečnosti nemusí být seznam *list* téhož typu, stačí, aby jeho typ byl nadtřídou typu pole *array*. To se může jevit jako velmi matoucí, ovšem pouze do té chvíle, dokud si neuvědomíme, že pokud máme např. pole celých čísel, tj. *Integer* a seznam obecných čísel *Number*, pak platí, že pole prvků typu *Integer* JE polem prvků typu *Number*! Skutečně, zde se dostáváme zpět ke klasické dědičnosti a nesmí nás mást pravidla, která platí pro obecné typy ve třídách.

Generics metody vs. wildcards

Jak již bylo zmíněno, je žádoucí, aby typ použitý u generické metody spojoval alespoň dva parametry nebo parametr a návratovou hodnotu. Následující příklad demonstruje nesprávné použití generické metody.

```
public static <T, S extends T> void copy(List<T> destination, List<S> source)
```

Příklad je syntakticky bezproblémový, dokonce jej lze i přeložit a bude fungovat dle očekávání. Nicméně správný zápis by měl být následující.

```
public static <T> void copy(List<T> destination, List<? extends T> source);
```

Zde je již vidět požadovaná vlastnost -- T spojuje dva parametry metody a přebytečné S je nahrazené žolíkem. V prvním příkladu si všimněte zápisu $S \text{ extends } T$. Ukazuje další možnou deklaraci generics.

Pole

Při deklaraci pole nelze použít parametrizovanou třídu, pouze třídu s žolíkem, který není vázaný (nebo bez použití žolíku). Tj. jediná správná deklarace je následující:

```
List<?>[] pole = new List<?>[10];
```

Parametrizovanou třídu v seznamu nelze použít z toho důvodu, že při vkládání prvků do nich runtime systém kontroluje pouze *typ* vkládaného prvku, nikoliv už to, zda využívá generics a zda tento odpovídá deklarovanému typu. To znamená, že měli bychom například pole seznamů, které obsahují pouze řetězce, mohli bychom do něj bez problémů vložit pole čísel. To by samo o sobě nic nezpůsobilo, ovšem mohlo by dojít k „přeměně“ typu generics, čímž by se seznam čísel „proměnil“ na seznam řetězců, což by bylo špatně.

Vícenásobná vazba generics

Uvažujme následující metodu (bez použití generics), která vyhledává maximální prvek nějaké kolekce. Navíc platí, že prvky kolekce musí implementovat rozhraní `Comparable`, což, jak lze snadno nahlédnout, není syntaxí vůbec podchyceno a tudíž zavolání této metody může vyvolat výjimku `ClassCastException`.

```
public static Object max(Collection c);
```

Nyní se pokusíme vymyslet, jak zapsat tuto metodu za použití generics. Chceme, aby prvky kolekce implementovali rozhraní `Comparable`. Podíváme-li se na toto rozhraní, zjistíme, že je též parametrizované generics. Potřebujeme tedy takovou instanci, která je schopná porovnat libovolné třídy v hierarchii nad třídou, která bude prvkem vstupní kolekce. První pokus, jak zapsat požadované.

```
public static <T extends Comparable<? super T>> T max(Collection<T> c);
```

Tento zápis je relativně OK. Metoda správně vrátí proměnnou stejného typu, jaký je prvkem v kolekci, dokonce i použití `Comparable` je správné. Nicméně, pokud bychom se zajímali o signaturu metody po „výmazu“ generics, dostaneme následující.

```
public static Comparable max(Collection c);
```

To neodpovídá signatuře metody výše. Využijeme tedy vícenásobné vazby.

1

```
public static <T extends Object & Comparable<? super T>> T max (Collection<T>
```

Nyní, po „výmazu“ má již metoda správnou signaturu, protože v úvahu se bere první zmíněná třída. Obecně lze použít více vazeb pro generics, například chceme-li, aby obecný prvek byl implementací více rozhraní.

Závěr

V článku jsme se seznámili se základními i některými pokročilými technikami použití *generics*. Tato technologie má i další využití, například u reflexe. Tohle však již překračuje rámec začátečnického se-

¹V materiálu, ze kterého čerpám, je navíc `Collection<? extends T>`. Domnívám se ovšem, že metoda zmíněná v tomto článku má stejnou funkcionalitu. Pokud se někomu podaří nalézt protipříklad, budu rád.

znamování s Javou.

Celý článek vychází z materiálů, které jsou volně k dispozici na oficiálních stránkách Javy firmy Sun [<http://java.sun.com/>], zejména pak z *Generics in the Java Programming Language* od Gilada Brachy. Některé příklady v této stati jsou převzaty ze zmíněného článku.