Dynamické datové struktury (kontejnery)

November 12, 2014

1 Dynamické datové struktury (kontejnery)

1.1 Kontejnery

- dynamické datové struktury vhodné k ukládání proměnného počtu objektů (přesněji odkazů na objekty)
- podle povahy kontejneru mohou mít různé specifické vlastnosti:

  seznamy každý prvek v nich uložený má svou pozici (podobně jako u pole, ale prvků lze uložit potenciálně neomezený počet)

  množiny prvek lze do množiny vložit nejvýš jedenkrát (odpovídá matematické představě), při porovnávání rozhoduje rovnost podle equals

  mapy v kontejneru jsou dvojice (klíč, hodnota), při znalosti klíče lze v kontejneru rychle najít hodnotu (přibližně odpovídá databázovému klíči)

- kontejnery jsou datové struktury v operační paměti, nejsou automaticky ukládané do trvalé paměti (např. na disk)

1.2 Kontejnery - k čemu slouží

- Kontejnery slouží k ukládání objektů (ne hodnot primitivních typů!) Při ukládání hodnot jako jsou čísla, booleovské hodnoty, znaky apod. se de facto ukládají jejich objektové protějšky, tzn. např. `Integer, Char, Boolean, Double` apod.
- V Javě byly původně koncipovány jako *beztypové* - bylo možno do jednoho kontejneru ukládat vždy objekty různých typů. To už ale od verze Java 5 neplatí.
- Od Javy 5 mají tedy kolekce tzv. *typové parametry* vyznačené ve špičatých závorkách (např. `List<Person>`), jimiž určujeme, jaké položky se do kolekce smějí dostat

1.3 Kontejnery – proč vůbec

- K. jsou dynamickými alternativami k poli a mají daleko širší použití
- Slouží k uchování proměnného počtu objektů -

- počet prvků se v průběhu existence kontejneru může měnit
- oproti polím nabízejí časově efektivnější algoritmy přístupu k prvkům

1.4 Kontejnery – co použít

Většinou se používají kontejnery hotové, vestavěné, tj. ty, jež jsou součastí Java Core API:

- vestavěné kontejnerové třídy jsou definovány v balíku `java.util`
- je možné vytvořit si vlastní implementace, obvykle ale zachovávající/implementující *standardní* rozhraní

1.5 Základní kategorie kontejnerů

Kategorie jsou dány tím, které rozhraní příslušný kontejner implementuje (základní jsou List¡E¿ (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/List.html`), Set¡E¿ (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/Set.html`) a Map¡K, V¿ (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/Map.html`)).

Seznam (List) lineární struktura, každý prvek má svůj číselný index (pozici)

Množina (Set) struktura bez duplicitních hodnot a obecně také bez uspořádání, umožňuje rychlé dotazování na přítomnost prvku

Asociativní pole, mapa (Map) struktura uchovávající dvojice (klíč→hodnota), rychlý přístup přes klíč

1.6 Kolekce

- jsou kontejnery implementující rozhraní *Collection* - API doc k rozhr. Collection (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/Collection.html`)
- Rozhraní kolekce popisuje velmi obecný kontejner, disponující operacemi: *přidávání, rušení prvku, získání iterátoru, zjišťování prázdnosti* atd.
- Mezi kolekce patří mimo *Map*y všechny ostatní vestavěné kontejnery - *List*, *Set*
- Prvky kolekce nemusí mít svou pozici danou indexem - viz např. *Set*

1.7 Kontejnery – rozhraní, nepovinné metody

- Funkcionalita vestavěných kontejnerů je obvykle předepsána výhradně *rozhraním*, jenž implementují.
- Rozhraní však připouštějí, že některé metody jsou *nepovinné*, třídy jej nemusí implementovat!
- Fakticky to v takovém případě vypadá tak, že metoda tam sice je, jinak by to typově nesouhlasilo, ale nelze ji použít, volání obvykle *vyhodí výjimku*.

- V praxi se totiž někdy nehodí implementovat jak čtecí, tak i zápisové operace, protože některé kontejnery chceme mít read-only

1.8 Iterátory

Iterátory jsou prostředkem, jak sekvenčně procházet prvky kolekce buďto

- v neurčeném pořadí nebo
- v uspořádání (u uspořádaných kolekcí)

Každý iterátor musí implementovat velmi jednoduché rozhraní Iterator¡E¿ (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/Iterator.html`) se třemi metodami:

- *boolean hasNext()*
- *E next()*
- *void remove()*

2 Generické typy. Typové parametry

2.1 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.

2.2 Seznam bez generických typů

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, což se používalo před Javou 5, kde jiná možnost ani nebyla:

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

2.3 Deklarace seznamu s generiky

Takhle s nimi pracujeme nyní:

```
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. Toto písmeno by zároveň mělo vystihovat použití resp. význam takového zobecnění. Tedy T je typ, E je prvek (element) a tak podobně

2.4 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 funkcionalita by byla pryč.

2.5 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);
```

2.6 Jednoduché využití v metodách (2)

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 funkcionalita by byla pryč.

2.7 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>();
```

2.8 První příklad použití

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);
```

2.9 První příklad použití

Jak vidíme v horním příkladu, do seznamu lze vložit libovolný objekt (byť zde jsme měli štěstí 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).

2.10 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);
}
```

2.11 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:

2.12 Žolíci (wildcards) (2)

```
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 čísly.

2.13 Žolíci (wildcards) (3)

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.*

2.14 Žolíci (wildcards) (4)

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 pomiňme 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;
}
```

2.15 Žolíci (wildcards) (5)

Opět, metoda se jeví jako bezproblémová. Nic ale není tak jednoduché, jak by se mohlo zdát. Nyní zkusíme 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 objeků 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 čísly*. Nyní je již pouze třeba ozřejmit syntaxi takové úvahy.

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

2.16 Žolíci (wildcards) (6)

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.

2.17 Žolíci (wildcards) (7)

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 setřídit. Jak lze taková čísla třídit? 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 třídit obecná celá čísla. Následující příklad demonstruje syntaxi a použití popsané konstrukce

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

2.18 Žolíci (wildcards) (8)

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 setříděné 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);
```

2.19 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 přenese prvky z pole nějakého typu přidá hodnoty do seznamu s prvky téhož typu.

2.20 Žolíci (wildcards) (10)

```
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.

2.21 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.

2.22 Žolíci (wildcards) (11)

```
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 extends T . Ukazuje další možnou deklaraci generics.

2.23 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.

2.24 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);
```

2.25 Vícenásobná vazba generics (2)

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);
```

2.26 Vícenásobná vazba generics (3)

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.V 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.

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

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.

2.27 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 seznamování s Javou. Celý článek vychází z materiálů, které jsou volně k disposici 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.

3 Seznamy

3.1 Seznamy

- lineární struktury
- implementují rozhraní *List* (rozšíření *Collection*) API doc k rozhr. List (`http://java.sun.com/javase/8/docs/api/java/util/List.html`)
- prvky lze adresovat indexem (typu int)
- poskytují možnost získat dopředný i zpětný *iterátor*
- lze pracovat i s *podseznamy*

4 Množiny

4.1 Množiny

Množiny

- jsou struktury standardně bez uspořádání prvků (ale existují i uspořádané, viz dále)
- implementují rozhraní *Set* (což je rozšíření *Collection*)

Cílem množin je mít možnost rychle (se složitostí O(log(n))) provádět atomické operace:

- vkládání prvku (*add*)
- odebírání prvku (*remove*)

- dotaz na přítomnost prvku (*contains*)
- lze testovat i relaci *je podmnožinou*

Standardní implementace množiny:

- *hašovací tabulka* (*HashSet*) nebo
- *vyhledávací strom* (černobílý strom, Red-Black Tree - *TreeSet*)

4.2 Množiny - implementace

Standardní implementace množiny:

hašovací tabulky HashSet (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/HashSet.html`) - potenciálně rychlejší, ale neuspořádané hodnoty

černobílé stromy TreeSet (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/TreeSet.html`) - uspořádané hodnoty, s garantovanou logaritmickou složitostí

4.3 Uspořádané množiny

Uspořádané množiny:

- Implementují rozhraní *SortedSet* - API doc k rozhraní SortedSet (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/SortedSet.html`)
- Jednotlivé prvky lze tedy iterátorem procházet v přesně definovaném pořadí - uspořádání podle *hodnot prvků*.
- Existuje vestavěná implementace *TreeSet* - černobílé stromy (Red-Black Trees) API doc ke třídě TreeSet (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/TreeSet.html`)

Uspořádání je dáno buďto:

- standardním chováním metody *compareTo* vkládaných objektů - pokud implementují rozhraní *Comparable*
- nebo je možné uspořádání definovat pomocí tzv. *komparátoru* (objektu impl. rozhraní *Comparator*) poskytnutých při vytvoření množiny.

5 Mapy

5.1 Mapy

Mapy (*asociativní pole*, nepřesně také hašovací tabulky nebo haše) fungují v podstatě na stejných principech a požadavcích jako *Set*:

- Ukládají ovšem dvojice (klíč, hodnota) a umožnují rychlé vyhledání dvojice podle hodnoty klíče.

- Základními metodami jsou: dotazy na přítomnost klíče v mapě (*containsKey*),
- výběr hodnoty odpovídající zadanému klíči (*get*),
- možnost získat zvlášť *množiny klíčů*, *hodnot* nebo *dvojic* (klíč, hodnota).

5.2 Mapy - implementace a složitosti

Mapy mají podobné implementace jako množiny, tj.:

hašovací tabulky HashMap (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/HashMap.html`) - potenciálně rychlejší, ale neuspořádané klíče

černobílé stromy TreeMap (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/TreeMap.html`) - uspořádané klíče, s garantovanou logaritmickou složitostí

Složitost základních operací (*put*, *remove*, *containsKey*):

- Mapy implementované jako stromy mají nejvýš logaritmickou složitost základních operací.
- U map implementovaných hašovací tabulkou složitost v praxi závisí na kvalitě hašovací funkce (metody hashCode) na ukládáných objektech, teoreticky se blíží složitosti konstantní.

5.3 Uspořádané mapy

Uspořádané mapy:

- Implementují rozhraní *SortedMap* - API doc k rozhraní SortedMap (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/SortedMap.html`)
- Dvojice (klíč, hodnota) jsou v nich *uspořádané podle hodnot klíče*.
- Existuje vestavěná implementace *TreeMap* - černobílé stromy (Red-Black Trees) - API doc ke třídě TreeMap (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/TreeMap.html`)
- Uspořádání lze ovlivnit naprosto stejným postupem jako u uspořádané množiny.

6 Funkcionální přístup v Javě

6.1 Funkcionální přístup v Javě

- Velmi malá část skutečného funkcionálního jazyka je obsažena v Javě 8.
- Jde o tzv. lambda výrazy (Lambda expressions).
- Umožňují syntakticky elegantní hutný zápis pomocí symbolu šipky, např. x $\rightarrow$ x+1.

- V zásadě bylo i dříve možné realizovat pomocí rozhraní s jednou metodou, implementovaných obvykle anonymní vnitřní třídou.

6.2 Podpora v jazyce

- Syntaktický konstrukt →, např.
- `(Person p) $\to$ p.printInfo();`

6.3 Podpora v knihovnách (Java Core API)

- funckionální rozhraní (functional interfaces) (`http://docs.oracle.com/javase/specs/jls/se8/html/jls-9.html\#jls-9.8`) v balíku `java.util.functions`
- většinou jako generická (typově parametrizovaná) rozhraní, např.
- Predicate¡T¿ s jednou metodou boolean test(T t)
- Consumer¡T¿ s jednou metodou void accept(T t)
- Consumer¡T¿ s jednou metodou void accept(T t)

6.4 Dokumentace

- Oracle Java Documentation: Lambda Expressions (`http://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html`)

6.5 Příklad

- Předávání booleovských funkcí jako predikátů do metod.

7 Další informace a odkazy

7.1 Kontejnery - souběžný přístup, výjimky

- Moderní kontejnery jsou *nesynchronizované*, nepřipouštějí souběžný přístup z více vláken.
- Standardní, nesynchronizovaný, kontejner lze však zabalit synchronizovanou obálkou.
- Při práci s kontejnery může vzniknout řada *výjimek*, např. *IllegalStateException* apod.
- Většina má charakter výjimek *běhových*, není povinností je odchytávat - pokud věříme, že nevzniknou.

7.2 Starší typy kontejnerů, Enumeration

Existují tyto starší typy kontejnerů (za → uvádíme náhradu):

- *Hashtable* → *HashMap*, *HashSet* (podle účelu)
- *Vector* → *List*
- *Stack* → *List*

Roli iterátoru plnil dříve *výčet* (Enumeration (`http://docs.oracle.com/javase/8/docs/api/index.html?java/util/Enumeration.html`)) se dvěma metodami:

- *boolean hasMoreElements()*
- *Object nextElement()*

7.3 Srovnání implementací kontejnerů

Seznamy:

- na bázi pole (*ArrayList*) - rychlý přímý přístup (přes index)
- na bázi lineárního zřetězeného seznamu (*LinkedList*) - rychlý sekvenční přístup (přes iterátor)

téměř vždy se používá *ArrayList* - stejně rychlý a paměťově efektivnější*Množiny a mapy*:

- na bázi hašovacích tabulek (*HashMap*, *HashSet*) - rychlejší, ale neuspořádané (lze získat iterátor procházející klíče uspořádaně)
- na bázi vyhledávacích stromů (*TreeMap*, *TreeSet*) - pomalejší, ale uspořádané
- spojení výhod obou - *LinkedHashSet*, *LinkedHashMap*

7.4 Odkazy

Demo efektivity práce kontejnerů - Demo kolekcí (`http://www.fi.muni.cz/~tomp/java/ucebnice/javasrc/tomp/ucebnice/kolekce/Kolekce.java`) (bez-typových, nepoužívajících generické typy)Velmi podrobné a kvalitní seznámení s kontejnery najdete na Trail: Collections (`http://docs.oracle.com/javase/tutorial/collections/`)