Kolekcie

Už z úvodných prednášok vieme, že do premenných vieme ukladať informáciu konkrétneho typu, napríklad reťazce, čísla, ale aj referencie na objekty (filmy, ľudí atď.). Veľmi často sa však stáva, že chceme pracovať s viacerými objektami naraz, alebo mať premennú, do ktorej vieme uložiť skupinu objektov.

To zodpovedá aj príkladom zo života, veď máme kolekciu filmov, zoznam ľudí, ktorých chceme pozvať na narodeninovú párty alebo nudnú matematickú množinu párnych čísiel.

Polia

Samozrejme, Java nezaostáva voči programovacím jazykom, ktoré poskytujú podporu pre premenné obsahujúce skupiny dát. Jeden zo spôsobov, ktorý sme používali už pri práci s korytnačkami, je používanie **polí**.

Kým premennú si môžeme predstaviť ako „chlievik“ v pamäti, do ktorého vieme uložiť *jeden* objekt (alebo primitívny typ), pole predstavuje spôsob, ktorým sa možno odkazovať na viacero „chlievikov“ vedľa seba.

Deklarácia poľa

Spomeňme si na deklaráciu prázdneho poľa reťazcov a neprázdneho poľa čísiel, ktoré hneď naplníme dátami.

```
// pole dĺžky tri, ktoré môže obsahovať až tri mená
String[] mená = new String[3];
// pole čísiel dĺžky 5
int[] niektoréPárneČísla = new int[] {2, 4, 6, 8, 10};
```

Takisto môžeme vytvoriť napríklad pole filmov:

```
Film matrix = new Film(...);
Film casablanca = new Film(...);
Film[] mojeObľúbenéFilmy = new Film[2];
mojeObľúbenéFilmy[0] = matrix;
mojeObľúbenéFilmy[1] = casablanca;
```

Limity práce s poľami

Polia v Jave však majú viacero nevýhod: predovšetkým **pevnú dĺžku**. Ak na začiatku vytvoríme pole dĺžky *X*, nemôžeme ho predĺžiť, ani skrátiť. Pridanie nového filmu do dvojprvkového poľa **mojeObľúbenéFilmy** z príkladu nie je možné: museli by sme vytvoriť nové pole dĺžky tri, skopírovať doň prvky z pôvodného poľa a na koniec vložiť nový prvok.

Tým sa dostávame k ďalšej nevýhode: nemožnosti efektívne vkladať prvky, ani mazať prvky v poli. Vkladanie i mazanie vyžaduje vytváranie nového poľa a prehadzovanie prvkov z pôvodného poľa s prípadným vynechaním zmazaných, resp. vkladaním nových prvkov.

Práca s takýmito prvkami je potom neefektívna a často nastávajú rôzne chyby (napríklad klasická chyba „plus-mínus 1“, kde sa pomýlime o jeden index).

Kolekcie

Našťastie, v súčasnej Jave sa s poľami musíme zaoberať len minimálne. K dispozícii je totiž sada tried pre **kolekcie**, teda sady objektov. Za kolekciu považujeme všetky dátové typy, ktoré pracujú so skupinami objektov. Zoznamy, množiny, či dokonca mapy.

V ďalších sekciách sa každej kolekcii budeme venovať samostatne a ukážeme si vlastnosti a príklady použitia.

Zoznamy

Zoznamy bývajú najčastejšie používanou dátovou štruktúrou, a to hlavne preto, že do veľkej miery suplujú úlohu polí – ale odstraňujú takmer všetky nepríjemnosti, s ktorými je spätá práca s nimi.

Pri práci s poľami vieme, že každé pole musí mať špecifikovaný dátový typ prvkov, ktoré sa v nich nachádzajú. Máme teda **pole reťazcov**, **pole filmov**, atď. To isté platí aj pre zoznamy: pri zozname musíme takisto definovať dátový typ prvkov, ktoré sa v ňom nachádzajú.

Ukážme si jednoduchý príklad:

Vytvorenie zoznamu niekoľkých prvkov

```
List<String> mená = Arrays.asList("Ringo", "George“, "Terry", "Michael“);
```

Volaním (statickej) metódy **asList()** na triede **java.util.Arrays**, kde do parametrov uvedieme ľubovoľný počet prvkov, vieme vytvoriť zoznam, presnejšie objekt typu **java.util.List**.

Prechod zoznamom cez for-each

Tento zoznam vieme elegantne prechádzať pomocou skráteného **for-each** cyklu:

```
for(String meno : mená) {
  System.out.println(meno);
}
```

Prechod zoznamom klasickým cyklom

Tento cyklus je ekvivalentný nasledovnému zápisu:

```
for(int i = 0; i < mená.size(); i++) {
  String meno = mená.get(i);
  System.out.println(meno);
}
```

Tento spôsob je veľmi podobný prechádzaniu polí, ale je v ňom niekoľko syntaktických odlišností.

Dĺžka zoznamu

Iste si pamätáme, že dĺžku poľa **poleMien** vieme zistiť pomocou inštančnej premennej **length**. Na rozdiel od nich vieme zistiť dĺžku zoznamu pomocou metódy **size()**.

Získanie prvku zo zoznamu

Prvky z poľa sa získavali pomocou hranatých zátvoriek, napríklad tretí prvok poľa mien vieme získavať cez **poleMien[2]** (polia sú indexované od nuly). Zoznamy sú však plnoprávne objekty s metódami, a získať *i*-ty prvok vieme získať pomocou metódy **get()**.

Všimnime si ešte jednu dôležitú vec: trieda **java.util.List** je interfejsom, z ktorého možno odvodiť zoznam všetkých metód (operácií), ktoré poskytuje zoznam.

V prípade metódy **Arrays.asList()** platí ešte jedna drobnosť: zoznam, ktorý je výsledkom volania, má pevnú dĺžku a nemožno doň ani pridávať prvky, ani odoberať (možno ich však meniť).

Pridanie prvku do zoznamu

Ako však pridať prvok? V prípade polí sme spomínali, že pridávanie je komplikované. To však v zoznamoch našťastie neplatí, pretože máme k dispozícii metódu **add()**, ktorá spraví všetku čiernu robotu za nás.

ArrayList

Ukážme si to na príklade, a dokonca príklade konkrétnej implementácie interfejsu **List**. Najčastejšie používanou implementáciou je **java.util.ArrayList**, ktorá zodpovedá zoznamu pracujúceho nad poľom. Ako vytvoriť jeho inštanciu? Jednoducho:

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

Netreba sa zľaknúť komplikovanej syntaxe plnej zátvoriek. Stačí si uvedomiť, že plná špecifikácia triedy je „zoznam reťazcov“, teda **ArrayList<String>**. Guľaté zátvorky zodpovedajú volaniu konštruktora. Ak chceme vytvárať nový film, zavoláme **new Film()**. Ak chceme vytvoriť novú inštanciu zoznamu reťazcov, vytvárame ju cez konštrukciu **new ArrayList<String>()**.

Interfejs na ľavej strane zodpovedá filozofii z minulej prednášky. Používateľa objektu **mená** nezaujímajú krvavé detaily v útrobách **ArrayList**u, dôležité je, že má objekt, ktorý poskytuje metódy pre prácu so všeobecným zoznamom.

Vyššie uvedenou konštrukciou vytvoríme prázdny zoznam, teda zoznam dĺžky nula.

Pridávanie prvku do zoznamu

Pridať prvok do tohto zoznamu je tiež jednoduché:

```
mená.add("Ringo“)
```

Pridaním prvku sa zoznam natiahne na dĺžku jedna. Čo sa deje vo vnútri? To nás v prvej fáze zaujímať nemusí. Rozhodne sa nemusíme starať o nejaké naťahovanie polí, prehadzovanie prvkov, atď. Trieda **ArrayList** toto všetko rieši za nás.

A naozaj, ak by sme si otvorili zdrojový kód tejto triedy, videli by sme, že sa v nej vehementne pracuje s poľom objektov. Pridanie prvkov vytvorí nové pole, a poprehadzuje doň dáta zo starého poľa, odobratie prvku zase urobí podobnú vec, ibaže neželaný prvok vynechá. Už len názov triedy indikuje, že zoznam bude mať niečo do činenia s poľom (*array* = pole).

Tu sa ukazuje prvá výhoda zoznamu (nielen **ArrayList**) : zoznam „dýcha“ – ak doňho pridáme prvok, nafúkne sa, a ak prvok odoberieme, zoznam sa „sfúkne“. Miesta v zozname máme dosť: vieme doňho dať 2^{32} prvkov (cca 2 miliardy).

Pridanie prvku na pozíciu

V prípade **ArrayListu** platí, že prvky sa automaticky pridávajú na koniec zoznamu. Ak chceme pridať prvok na začiatok, využijeme prekrytú metódu **add(index, novýPrvok)**:

```
mená.add(0, "George“)
```

Tento kód pridá Georga na začiatok zoznamu. Treba si len dať pozor na to, aby index nebol mimo rozsahu zoznamu (záporný, či väčší alebo rovný dĺžke zoznamu), v opačnom prípade nastane výnimka **IndexOutOfBoundsException**.

Ak pridáme do zoznamu nový prvok, automaticky zvýšime indexy prvkov napravo od vkladaného prvku o jedna.

0	1	2
Marek	Matúš	Ján

Po vložení prvku cez **add(1, "Lukáš“)** získame:

0	1	2	3
Marek	Lukáš	Matúš	Ján

Zmena hodnoty prvku

Čo v prípade, že chceme zmeniť hodnotu prvku s daným indexom? Môžeme využiť metódu **set()**.

```
List<String> mená = Arrays.asList(“George”, “Tim”, “Peter”);
mená.set(1, “John”); // zmení Tim na John
```

Podotknime, že index v metóde musí spadať do rozsahu zoznamu (interval 0 až dĺžka zoznamu - 1).

Odstránenie prvku podľa indexu

Pridávať teda už vieme, ale ako odstraňovať prvky? Pomocou metódy **remove()**, ktorej parametrom je index prvku, ktorý chceme odstrániť. V nasledovnom príklade odstránime tretí prvok zoznamu (teda z indexu 2):

```
mená.remove(2);
```

Všetky prvky napravo od odstraňovaného prvku sa posunú doľava a ich indexy sa znížia o jedna.

Po odstránení cez **mená.remove(1)**:

0	1	2
George	Tim	Peter

Tým získame dvojprvkový zoznam:

0	2
George	Peter

V prípade, že index je mimo rozsahu zoznamu (záporný, či väčší alebo rovný dĺžke zoznamu), nastane výnimka **IndexOutOfBoundsException**.

Odstránenie prvku

Zoznam poskytuje možnosť odstraňovania konkrétnych objektov. Odstrániť *Georga* zo zoznamu môžeme nasledovne:

```
mená.remove("George“);
```

V prípade, že sa v zozname nachádza viac *Georgov*, odstráni prvok s najmenším indexom.

Odstraňovanie prvkov očakáva, že trieda prvku má v sebe exaktne definovanú metódu **equals()**. V opačnom prípade môže byť správanie tejto metódy nepredvídateľné. Diskusia o metóde **equals()** sa nachádza v stati o množinách nižšie.

Konštruktory zoznamu

Zoznam **ArrayList** poskytuje výhodnú možnosť vytvoriť sa na základe existujúcej inštancie. To možno využiť v prípade, že chceme vyrobiť kópiu zoznamu. Nezabúdajme na to, že ak máme prípad:

```
List<String> mená = Arrays.asList("George", "Tim", "Peter");
List<String> novéMená = mená;
```

tak **novéMená** a **mená** sú referenciou na ten istý zoznam. Ak odstránime prvok zo zoznamu **mená**, zmena sa prejaví aj v **novýchMenách**. To môže viesť k potenciálnym chybám, ktoré sa ťažko ladia.

Ak chceme vytvoriť kópiu zoznamu, vieme využiť konštruktor:

```
List<String> mená = Arrays.asList("George", "Tim", "Peter");
List<String> kópiaMien = new ArrayList<String>(mená);
```

Zoznamy primitívnych typov

V príklade sme zatiaľ používali zoznam reťazcov. Čo v prípade, že chceme vytvoriť zoznam čísiel? Jednou možnosťou je použiť **Arrays.asList()**

Aký dátový typ bude mať zoznam celých čísiel? Prvá vec, čo nám napadne, bude využiť znalosti z polí:

```
List<int> niektoréPárneČísla = Arrays.asList(2, 4, 6, 9. 10);
```

V tomto prípade však narazíme na syntaktickú chybu.

Zdôvodnenie je prosté: Java nepodporuje zoznamy primitívnych typov (**int** je primitívny typ). Znamená to azda, že nemôžeme mať zoznam čísiel? To rozhodne nie!

Toto obmedzenie možno ľahko obísť, potrebujeme na to však poznať niektoré technické detaily.

Musíme však urobiť drobný lyrický exkurz.

Autoboxing

Vieme, že všetky premenné v Jave sa rozdeľujú do dvoch skupín: primitívne typy (**int**, **boolean**...). a na objektové typy. Primitívne typy sú výpočtovo a pamäťovo veľmi efektívne, ale majú jednu nevýhodu: nemôžeme na nich volať žiadne metódy. Rozpoznať ich možno podľa toho, že ich názov sa začína malým písmenom. Objektové typy sú všetky ostatné dátové typy založené na triedach.

Takéto striktné delenie má výhodu vzhľadom k spomínanej efektivite, na druhej strane komplikuje prácu s primitívmi v niektorých špecifických situáciách a čo je horšie, narúša čistotu objektovo orientovanej filozofie.

Wrappery

Premostiť výhody primitívnych a objektových typov Java rieši pomocou tzv. obaľovačov (*wrapper*) primitívnych typov. V samej podstate ide o jednoduchú vec: každý primitívny typ má svoj obaľovač, teda objektový protipól:

Primitívny typ	Obaľovač (Wrapper) založený na objekte
int	java.lang.Integer
boolean	java.lang.Boolean
long	java.lang.Long
double	java.lang.Double
...	...

Inak povedané, každý primitívny typ má svojho kamaráta, ktorého meno sa začína veľkým písmenom.

Prevod medzi primitívnymi typmi a objektovými wrappermi je našťastie jednoduchý, pretože Java poskytuje vlastnosť zvanú **autoboxing** (*automatické škatuľkovanie*), ktorou sa automaticky dokážu previesť primitívne typy na wrappery a naopak.

Dajme si jednoduchý príklad:

```
Integer dva = 2;
```

Do premennej typu **Integer** (objektového typu) vložíme primitívny typ. Autoboxing automaticky zaistí prevod, inak povedané, odtieni nás od písania nasledovného kódu:

```
Integer dva = new Integer(2);
```

To funguje aj naopak:

```
Integer objektOsemnásť = new Integer(18);
int osemnásť = objektOsemnásť;
```

Na druhom riadku prebehol autoboxing: do premennej primitívneho typu **int** sa vložil obsah objektu **Integer**.

Ďalšou netypickou vlastnosťou je, že vďaka autoboxingu vieme robiť matematické operácie nad objektovými wrappermi bez nejakých špeciálnych ručných konverzií.

```
Integer objektOsemnásť = new Integer(18);
Integer objektDva = new Integer(2);
int dvadsať = objektOsemnásť + objektDva;
```

Toto takpovediac narúša striktnosť Javy, pretože za normálnych okolností môžeme cez **+** sčítavať len primitívne typy. Vďaka autoboxingu však vieme sčítavat (odčítavať, atď) aj objektové wrappery.

Opäť zoznamy primitívnych typov

Teraz, keď vieme používať autoboxing, je vytvorenie zoznamu čísiel jednoduchou záležitosťou. V dátovom type prvkov zoznamu uvedieme objektový wrapper.

```
List<Integer> niektoréPárneČísla = Arrays.asList(2, 4, 6, 8, 10);
```

Vyťahovať prvky z tohto zoznamu je prosté, používame bežné metódy **get**() a ich výsledky môžeme priraďovať do premenných primitívnych typov:

```
int prvéPárneČíslo = niektoréPárneČísla.get(0);
```

Vyššie uvedený zápis využíva autoboxing – bez neho by sme to museli robiť takto:

```
Integer prvéPárneČísloAkoObjekt = niektoréPárneČísla.get(0);
int prvéPárneČíslo = prvéPárneČísloAkoObjekt.intValue();
```

Ak chceme vytvoriť meniteľný zoznam primitívnych typov, napr. reálnych čísiel, nie je nič jednoduchšie: stačí vytvoriť inštanciu **ArrayList**u a sme v suchu.

```
List<Double> teplotyZaVíkend = new ArrayList<Double>();
teplotyZaVíkend.add(15.5);
teplotyZaVíkend.add(9.5);
```

Zoznamy objektov

Zmienili sme sa o zoznamoch reťazcov i o zoznamoch primitívnych typov. Čo však so zoznamami objektov? Našťastie, práca so zoznamami objektov sa ničím nelíši od práce s reťazcami (veď koniec koncov, aj reťazec je objekt). Stačí uviesť do lomených zátvoriek dátový typ a sme vybavení.

```
Film matrix = new Film(...);
Film casablanca = new Film(...);

List<Film> mojeObľúbenéFilmy = new ArrayList<Film>();
mojeObľúbenéFilmy.add(matrix);
mojeObľúbenéFilmy.add(casablanca);

System.out.println("Počet obľúbených filmov: "
  + mojeObľúbenéFilmy.size());
```

Typické idiómy pre prácu so zoznamami

Pri práci so zoznamami sa veľmi často opakujú isté „ustálené konštrukcie“ (*idioms*). Plnia úlohu frazeologizmov v slovenčine, pretože bežný programátor sa nad nimi nezamýšľa, ale má ich zažité v krvi. Rozoberme si niektoré z nich.

Index prvku v zozname

Bežnou situáciou je snaha nájsť v zozname konkrétny prvok a zistiť jeho index.

```
List<String> mená = Arrays.asList("George", "Tim", "Peter");
String hľadanéMeno = "Tim";

int index = -1; // -1 = index nenájdeného prvku
for(int i = 0; i < mená.size(); i++) {
  String meno = mená.get(i);
  if(hľadanéMeno.equals(meno)) {
    index = i;
  }
}
```

Tento idióm však možno skrátiť na použitie metódy **indexOf()**:

```
List<String> mená = Arrays.asList("George", "Tim", "Peter");
int index = mená.indexOf("Tim");
```

Táto metóda vráti index hľadaného prvku. V prípade, že je v zozname viacero rovnakých prvkov, vráti najmenší index. Ak sa prvok v zozname nenachádza, vráti -1.

Metóda **indexOf()** očakáva, že trieda prvku má v sebe exaktne definovanú metódu **equals()**. V opačnom prípade môže byť správanie tejto metódy nepredvídateľné. Diskusia o metóde **equals()** sa nachádza v stati o množinách nižšie.

Vyhľadanie najmenšieho prvku

Ak pracujeme so zoznamami čísiel, často je žiaduce nájsť najmenší prvok. Idea je prostá: prechádzame zoznamom a v pomocnej premennej si priebežne udržiavame najmenší prvok z časti zoznamu, ktorú sme už prešli. Na začiatku nastavíme minimum na dostatočne veľkú hodnotu, aby sme ju už v prvom kroku zmenšili.

```
List<Integer> čísla = Arrays.asList(10, -5, 3, -10);
int minimum = Integer.MAX_VALUE; // dve miliardy
for(int číslo : čísla) {
  if(číslo < minimum) {
    //aktuálne číslo je menšie než dosiaľ dosiahnuté minimum...
    minimum = číslo;
    // minimum zmenšíme
  }
}
```

Iterácia	Aktuálny prvok	Aktuálne minimum	Nové minimum
Pred cyklom	-	Integer. MAX_VALUE	Integer. MAX_VALUE
0.	10	Integer. MAX_VALUE	10
1.	-5	10	-5
2.	3	-5	-5
3.	-10	-5	-10

Tento idióm sa dá skrátiť použitím metódy **Collections.min()**

```
List<Integer> čísla = Arrays.asList(10, -5, 3, -10);
int minimum = Collections.min(čísla);
```

Metóda dokáže nájsť minimum zo zoznamu ľubovoľných prvkov. V takom prípade sa očakáva, že objekty majú definované prirodzené usporiadanie (implementujú interfejs **Comparable**). Alternatívne je možné dodať inštanciu komparátora.

Vyhľadanie najväčšieho prvku

Vyhľadanie najväčšieho prvku sa riadi podobnou filozofiou, používa však prevrátené podmienky.

```
List<Integer> čísla = Arrays.asList(10, -5, 3, -10);
int maximum = Integer.MIN_VALUE; // mínus dve miliardy
for(int číslo : čísla) {
  if(číslo > maximum) {
    //aktuálne číslo je väčšie než dosiaľ dosiahnuté maximum...
    maximum = číslo;
    // minimum zväčšíme
  }
}
```

Vyhľadanie prvku spĺňajúceho kritérium

Čo ak chceme v zozname nájsť prvok, ktorý spĺňa nejaké kritérium? Napríklad prvého človeka, ktorého meno sa začína na „T“? Prejdem zoznam prvkov a hľadám požadovaný prvok.

```
List<String> mená = Arrays.asList("George", "Tim", "Peter");
String prvéMenoZačínajúceT = null;
for(String meno : mená) {
  if(meno.startsWith("T")) {
    prvéMenoZačínajúceT = meno;
  }
}
```

Filtrovanie zoznamu

Mnohokrát je tiež vhodné prefiltrovať zoznam, teda vrátiť nový zoznam s prvkami starého zoznamu spĺňajúcimi nejaké kritérium. Idea spočíva vo vytvorení nového zoznamu, prechádzaní starého a „prehadzovaní prvkov vidlami“ do nového zoznamu. Vráťme zo zoznamu kladné čísla:

```
List<Integer> čísla = Arrays.asList(10, -5, 3, -10);
List<Integer> kladnéČísla = new ArrayList<Integer>();

for(int číslo : čísla) {
  if(číslo > 0) {
    kladnéČísla.add(číslo);
  }
}
```

Zoznamy a polia

Zoznamy a polia plnia v Jave takpovediac tú istú úlohu, teda reprezentujú skupinu viacerých prvkov, kde sa na jednotlivé prvky vieme odkazovať pomocou celočíselného indexu. Základným funkčným rozdielom je automatické „nafukovanie a sfukovanie“ zoznamu, teda jeho automatické prispôsobenie dĺžky k počtu prvkov, ktoré sú v ňom uložené.

V súčasnosti možno povedať, že v Jave neexistuje veľa situácií, keď je nutné uprednostniť použitie poľa pred zoznamom. Azda jedinou výnimkou sú situácie, keď narábame z matematickými štruktúrami, kde je výkon a efektivita naozaj dôležitá, a nepotrebujeme dynamickú zmenu veľkosti. Takýmito prípadmi sú napríklad veľkorozmerné matice (reprezentované ako dvojrozmerné polia) či reprezentácia bludísk, ktorých veľkosť sa v rámci behu programu nemení.

Niekedy však predsa len nastane situácia, keď potrebujeme previesť zoznam na pole a naopak. Praktickým prípadom býva použitie triedy **java.io.File**, pomocou ktorej chceme získať zoznam súborov a adresárov v danom adresári. Použijeme metódu **listFiles()**, ktorá vracia pole inštancií typu **File**.

```
File adresárProjektu = new File("c:/projekty/paz");
File[] súboryAAdresáre = adresárProjektu.listFiles();
```

Prevod poľa na zoznam

Ako zmeniť toto pole na zoznam súborov? Vieme použiť známu metódu **Arrays.asList(),** ktorej dáme do parametra konkrétne pole. Metóda vráti zoznam **List** prvkov, ktorých typ je zhodný s typom prvkov v poli.

```
List<File> zoznamSúborovAAdresárov = Arrays.asList(súboryAAdresáre);
```

Zoznam na pole

Opačný prevod je samozrejme tiež možný, hoci z hľadiska zápisu je o niečo komplikovanejší. Ukážme si to na príklade zoznamu mien.

```
List<String> zoznamMien = Arrays.asList("George", "John", "Peter");
String[] mená = new String[3];
mená = zoznamMien.toArray(mená);
```

Logika je nasledovná: vezmeme zoznam mien (dĺžky tri). Následne vytvoríme nové trojprvkové pole reťazcov, a použijeme ho ako parameter metódy **toArray()**. Inak povedané, táto metóda naplní uvedené pole **mená** prvkami zo zoznamu **zoznamMien**.

Túto syntax možno sprehľadniť, hoci obeťou za trochu úvahy:

```
List<String> zoznamMien = Arrays.asList("George", "John", "Peter");
String[] mená = zoznamMien.toArray(new String[0]);
```

Metóda **toArray()** funguje nasledovne: z dátového typu prvkov poľa v parametri určí typ prvkov vo výslednom zozname. Ak je pole v parametri kratšie ako zoznam, vytvorí sa nové pole požadovanej dĺžky a nakopírujú sa doňho prvky zo zoznamu.

Keďže naše pole v parametri má dĺžku nula, vo útrobách metódy **toArray()** sa vytvorí nové pole dĺžky tri, nakopírujú sa doň elementy zoznamu a toto pole sa vráti ako výsledok volania metódy.

Implementácie zoznamov

ArrayList, poľový zoznam

Vieme už, že zoznamy sú v Jave realizované jednak pomocou interfejsu **List** a jednak pomocou konkrétnych implementácií, ktoré hovoria, ako konkrétne sa ukladajú prvky zoznamu. Spomínali sme zatiaľ jedinú implementáciu, zoznam nad poľom, teda **ArrayList**. Táto implementácia je natoľko výhodná, že v podstate si s ňou mnoho vývojárov vystačí. (A potom môžete pokojne preskočiť túto stať). Ak sa však chcete dozvedieť o ďalších implementáciách, čítajte ďalej.

LinkedList, spájaný zoznam

Ďalšou dôležitou implementáciou je tzv. spájaný zoznam (**linked list**, niekde tiež spájaný zoznam). Kým zoznam nad poľom manipuluje s vnútornou inštanciou poľa, ktoré podľa potreby kopíruje do dlhšieho poľa (ak pridávame prvok) alebo do kratšieho poľa (ak prvky odoberáme), spojový zoznam si možno predstaviť ako reťaz prepojených prvkov.

Implementácia spojových zoznamov je často nutnosťou najmä v nízkoúrovňových jazykoch, pretože je to v podstate jedna z mála možností, ako efektívne pracovať so zoznamami neobmedzenej dĺžky a nezabrať pritom viac pamäte, než je potrebné.

Spojový zoznam si môžeme predstaviť ako reťaz balónikov, ktorú na jednom konci drží dievčatko. Všimnite si, že každý jeden balónik je naviazaný na predošlý a ďalší balónik.

V rámci implementácie máme prvok zoznamu reprezentovaný ako inštanciu triedy, ktorá v sebe nesie referenciu na nasledovný prvok (teda spojivo).

Sekvenčný prístup v spojových zoznamoch

Základnou vlastnosťou spojového zoznamu je nutnosť **sekvenčného prístupu**. Na rozdiel pod poľového zoznamu, kde sme priamo mohli pristúpiť k ľubovoľnému prvku, tuto takúto výhodu nemáme. Ak chceme zistiť farbu tretieho balónika, musíme vyjsť od dievčatka a postupne si „priťahovať" k sebe prvý (zelený) balónik, od ktorého si vieme pritiahnuť druhý (žltý) balónik a až od neho sa vieme dostať k tretiemu balóniku, ktorý je červený.

Toto sekvenčné traverzovanie zoznamu je očividne nevýhodou, lenže dramaticky zlepšuje pamäťovú efektívnosť zoznamu. Poľový zoznam totiž potrebuje buď dostatočne veľké pole, ktoré je sčasti prázdne (teda vyžadujeme viac pamäte, ktoré leží ladom) alebo primerane veľké pole, ktoré je však nutné v prípade potreby kopírovať do dlhšieho či kratšieho poľa, čo predlžuje dobu vkladania prvku.

Sekvenčný zoznam spotrebuje vždy len toľko pamäte, koľko je v ňom prvkov a teda neexistujú v ňom žiadne „prázdne" prvky.

Inak povedané, časová zložitosť pre prístup k prvku v poľovom zozname je **O(1)** (konštantná), prístup k prvku v spojovom zozname vyžaduje cyklus, kde postupne prechádzame o prvého prvku (dievčatka) zoznamom až dovtedy, kým nenájdeme požadovaný prvok. Zložitosť je teda závislá od počtu prvkov zoznamu, čiže je **O(n)**.

Na druhej strane, vkladanie prvku na koniec zoznamu je pamäťovo efektívne (nepotrebujeme žiadnu pomocnú dátovú štruktúru) a zložitosť je prinajhoršom **O(n)** (ak musíme prejsť celý zoznam + **O(1)** (vloženie prvku). Ak vkladáme prvok na začiatok zoznamu, zložitosť je dokonca **O(1)**, čiže konštantná.

LinkedList vs. ArrayList

Kedy zvoliť spojový zoznam **LinkedList** a kedy poľový **ArrayList**? Dokumentácia k Jave radí:

„Ak často pridávate prvky na začiatok zoznamu, alebo prechádzate zoznamom preto, aby ste zmazali niektorý prvok z jeho vnútra, mali by ste zvážiť použitie spojového zoznamu. Tieto operácie totiž požadujú len konštantný čas (na rozdiel od lineárneho času potrebného v poľovom zozname). To je však výraznou obeťou za výkonnosť, pretože prístup cez indexy vyžaduje v spojovom zozname lineárny čas, na rozdiel od konštantnej doby prístupu v poľovom zozname."

My len dodáme hrubú zásadu, že ak sa neviete rozhodnúť, použite štandardný **ArrayList**.

Vector, vektor

Ďalšou implementáciou interfejsu **List** je trieda **java.util.Vector**, teda vektor. Predstavuje „relikt" z prehistorických verzií Javy, v ktorej bola jedinou možnosťou, ako reprezentovať dynamické pole. Z hľadiska funkcionality poskytuje rovnaké možnosti ako **ArrayList** a jeho jedinou priamou výhodou je *synchronizovaný prístup*, čo znamená, že k inštancii **Vector**a môže naraz pristupovať viacero vlákien. (Vláknové programovanie je vysokopokročilým konceptom, ktorému sa nebudeme venovať.)

Opäť však platí hrubá zásada, že v prípade dilemy je lepšie zvoliť **ArrayList**, pretože i viacvláknovú podporu je možno dosiahnuť iným spôsobom.

Popri týchto troch implementáciách sú k dispozícii aj ďalšie triedy. V tomto texte sa im nebudeme venovať, pretože buď majú konkrétne, úzko špecifikované použitie (**CopyOnWriteArrayList**), alebo predstavujú zastaralú funkcionalitu.

Množiny

Vezmime si takýto príklad: máme niekoľko e-mailových adries, a chceme na ne poslať jeden mail. Samozrejme, každý e-mailový klient dokáže poslať jednu správu na viac adries: stačí ich zaradom uviesť do kolónky **Komu**.

Otázka znie: ako reprezentovať túto skupinu adries? Samozrejme, môžeme zvážiť použitie zoznamu alebo poľa. Ale ak sa pozrieme na sadu prijímateľov, zistíme dve odlišnosti. Predovšetkým, skupina neobsahuje viacero rovnakých adries, čo je prirodzené: asi nechceme, aby jeden prijímateľ dostal ten istý e-mail dvakrát. Ďalej nám vôbec nezáleží na poradí – mail posielame všetkým naraz a poradie adresátov nie je dôležité.

Takúto skupinu (kolekciu) objektov vieme pokojne reprezentovať množinou (angl. *set*). To zodpovedá aj matematickému chápaniu, v ktorom je množinou nazývaná skupina prvkov, v ktorej sa nenachádzajú rovnaké prvky, a v ktorej nezáleží na poradí.

Je zjavné, že množina {duro@abc.com, fero@abc.com, milan@server.sk} je rovná množine {fero@abc.com, milan@server.sk, duro@abc.com}. Ďalej vieme, že {duro@abc.com, duro@abc.com, duro@abc.com} nie je správne skonštruovaná množina, lebo platí, že {duro@abc.com}, do ktorej pridáme duro@abc.com, sa nezmení.

Set a HashSet

V Jave sú množiny reprezentované interfejsom **java.util.Set**, kde si môžeme zvoliť implementáciu **java.util.HashSet**.

Príklad na vytvorenie množiny e-mailových adries (ktoré reprezentujeme ako reťazce) je jednoduchý:

```
Set<String> adresáti = new HashSet<String>();
```

Prázdnosť množiny

Množina je na začiatku prázdna, čo vieme zistiť jednoducho pomocou metódy **isEmpty()** **(true** indikuje prázdnu množinu).

```
if(adresáti.isEmpty()) {
  System.out.println("Množina je prázdna.")
}
```

Pridávanie prvkov

Nič nebráni pridávať nové prvky pomocou známej metódy **add()**:

```
adresáti.add("duro@abc.com")
adresáti.add("fero@abc.com")
adresáti.add("milan@server.sk")
```

Počet prvkov

Metódou **size()** zistíme počet prvkov (mohutnosť množiny), teda v tomto prípade 3:

```
System.out.println(adresáti.size());
```

Čo sa stane, ak pridáme do množiny rovnaký prvok viackrát?

```
Set<String> adresáti = new HashSet<String();
adresáti.add("duro@abc.com")
adresáti.add("duro@abc.com")
```

Po prvom volaní metódy **add()** bude mohutnosť množiny 1. Po pridaní ďalšieho prvku sa mohutnosť nezmení. Ako je to dosiahnuté?

Rovnosť prvkov

Množiny pevne súvisia s pojmom rovnosť prvkov. Je to zaujímavý pojem, na ktorý sa možno dívať z matematického, ale i informatického, či filozofického hľadiska. Kedy vieme povedať, že sa dva objekty rovnajú?

Rovnosť reťazcov (objekty typu **String)** je v Jave definovaná v klasickom duchu: dva reťazce *R,* S sa rovnajú, ak sa rovnajú po znakoch, teda ak 1. znak reťazca *R* je zhodný s 1. znakom reťazca *S*, 2. znak *R* sa zhoduje s 2. znakom *S*, atď.

Z hľadiska implementácie je rovnosť riešená v rámci metódy **equals()**, ktorá je definovaná v triede **Object**. Každá trieda by mala prekrytím tejto metódy definovať podmienky rovnosti jej dvoch inštancií. V triede **String** je metóda **equals()** definovaná v duchu myšlienky z predošlého odseku. Potom vieme zistiť, či

```
String adresát1 = "jan@jan.com";
String adresát2 = "zuzana@spolocnost.com";
if(adresát1.equals(adresát2)) {
  // podmienka nebude splnená
}
```

Metóda **add()** sa teda vyslovene spolieha na to, že objekt pridávaný do množiny má v definícii svojej triedy zmysluplne prekrytú metódu **equals()**. V opačnom prípade je správanie množiny nedefinované a môže sa prejavovať náhodným chovaním!

Hašovacie kódy

S metódou **equals()** úzko súvisí metóda **hashCode()**. Hašovací kód sa snaží vystihnúť dáta objektu pomocou jediného čísla. Inak povedané, hašovací kód je výsledkom hašovacej funkcie, ktorá prevedie veľké množstvo dát do jediného celočíselného údaja.

Hašovacie kódy sa používajú hlavne na zrýchľovanie vyhľadávania v tabuľkových dátach alebo zoznamoch. Ak máte inštanciu triedy **Film** *Matrix* (s množstvom inštančných

premenných), a určíte jeho hašovací kód ako napr. 23268, môžete ho použiť ako index v zozname. Ak uložíte *Matrix* do indexu 23268, následné vyhľadanie tohto filmu v zozname sa zjednoduší. V rámci implementácie vyhľadávania v zozname už nemusíte prechádzať prvok za prvkom a zisťovať, či ste našli požadovaný film. Stačí vyrátať hašovací kód vyhľadávaného prvku, čím získate index v zozname, z ktorého vyhľadávaný prvok rovno vytiahnete.

Je však nutné pamätať na to, že hašovací kód nie je identifikátor objektu. Vzhľadom na to, že hašovacia funkcia „z množstva údajov vyrobí jeden údaj“, pokojne sa môže stať, že dva rozličné objekty budú mať rovnaký hašovací kód (nastane teda *kolízia*). Jednou z vlastností dobrej hašovacej funkcie je snaha o minimalizáciu kolízií. Ak v príklade zoznamu z filmov z predošlého odseku nastane kolízia, znamená to, že v políčku zoznamu s daným indexom budeme musieť udržiavať viacero objektov. Napriek tomu je však vyhľadávanie oveľa rýchlejšie ako v prípade prechádzania celého zoznamu.

Problematika hašovacích funkcií (a výber správnej funkcie vzhľadom k danému problému) je i v súčasnosti predmetom výskumu. V Jave existuje niekoľko typicky používaných hašovacích funkcií. Našťastie sa nimi nemusíme zaoberať, keďže každé rozumné vývojové prostredie poskytuje nástroj na vygenerovanie kódu, ktorý vyráta hašovací kód z danej inštancie.

Úzky súvis medzi hašovacím kódom a identitou je možno už zjavný. Má platiť zásada, že ak sú dva objekty rovnaké, musia mať rovnaký hašovací kód. (Naopak to platiť nemusí vzhľadom k vyššie uvedenej zásade o identifikátoroch.) V prípade, že to tak nie je, môže nastať nečakané chovanie v rozličných algoritmoch a triedach, ktoré na tomto predpoklade stavajú, a množina **Set** je jednou z nich.

Vždy, keď používate množiny, uistite sa, že máte explicitne definovanú metódu **hashCode()** a **equals()** na jej prvkoch. Objektové obaľovače primitívov, reťazce, a väčšina tried zabudovaných v Jave túto metódu majú definovanú. Do vašich vlastných tried budete túto metódu musieť dodať explicitne.
Eclipse vám pomôže pri generovaní týchto metód do tej miery, že sa nad nimi nemusíte priveľmi zamýšľať. V danej triede použite príkaz **Source | Generate hashCode() and equals()**, ktorý doplní kód tejto metódy automaticky.
Nezabudnite na to, že ak pridáte, či odoberiete inštančnú premennú v triede, tieto metódy musíte vygenerovať nanovo!

Podobne ako v prípade zoznamov, vieme vytvoriť množinu na základe akejkoľvej inej kolekcie. Stačí použiť konštruktor, ktorý berie ako parameter kolekciu, teda objekt typu **Collection**.

Vytvoriť množinu niektorých párnych čísiel možno nasledovne:

```
List<Integer> čísla = Arrays.asList(2, 4, 6, 8, 10);
Set<Integer> množinaČísiel = new HashSet<Integer>(čísla);
```

Prechádzanie množinou

Ako prechádzať prvkami množiny? Môžeme využiť výhody **for-each** cyklu:

```
for(String email : adresáti) {
  System.out.println(email);
}
```

Ak však použijeme klasický **for**, veľmi rýchlo narazíme na prekážku:

```
for(int i = 0; i < adresáti.size(); i++) {
  String email = adresáti.get(i);
  System.out.println(email);
}
```

Metóda **get()** v množine neexistuje!

Množina **Set** totiž nemá metódu **get()**! Hoci sa to môže zdať hrozné, zodpovedá to matematickej filozofii. Keďže množina nemá ustanovené poradie prvkov, nevieme jasne určiť poradie. Prvok duro@abc.com môže byť na prvom mieste, ale pokojne i na treťom, pretože v množine sa to vôbec neberie do úvahy.

Inak povedané, množina neumožňuje odkazovať sa na konkrétne prvky. Ak chceme vyhľadať nejaký konkrétny prvok, neostáva nič iné, len postupne prejsť prvkami množiny a zastaviť sa vtedy, ak ho nájdeme.

Príslušnosť prvku do množiny

Set nám však dáva možnosť zisťovať príslušnosť prvku do množiny, čo je tradičná matematická operácia ∈. Slúži na to metóda **contains()** s parametrom predstavujúcim prvok, o ktorom chceme zistiť príslušnosť do množiny.

```
if(adresáti.contains("duro@abc.com")) {
  System.out.println("Duro je v adresatoch.");
}
```

Táto metóda opäť stojí a padá na správnej definícii metód **hashCode()** a **equals()**.

Zjednotenie množín

S množinami sa dajú vykonávať všetky typické matematické operácie. Zjednotenie dvoch množín možno dosiahnuť pomocou metódy **addAll()**.

```
Set<String> kamaráti = new HashSet<String>();
kamaráti.add("peter@sidlisko.sk");
kamaráti.add("jana@sidlisko.sk");

Set<String> kolegovia = new HashSet<String>();
kolegovia.add("peter@firma.sk");
kolegovia.add("milan@firma.sk");
kolegovia.add("jana@sidlisko.sk");

Set<String> všetciAdresáti = new HashSet<String>();
všetciAdresáti.addAll(kamaráti);
všetciAdresáti.addAll(kolegovia);
```

V príklade sme teda urobili: VšetciAdresáti = Kamaráti ∪ Kolegovia. Všimnite si, že **jana@sidlisko.sk** sa vo výslednej množine zjaví len raz.

Parameter metódy **addAll()** je však natoľko flexibilný, že doň môžeme vložiť akúkoľvek inú kolekciu, teda aj zoznam.

```
Set<Integer> niektoréPárneČísla = new HashSet<String>();
niektoréPárneČísla.add(2);
niektoréPárneČísla.add(4);
niektoréPárneČísla.add(6);
List<Integer> ďalšiePárneČísla = Arrays.asList(8, 9, 10);
niektoréPárneČísla.addAll(ďalšiePárneČísla);
```

Prienik množín

Prienik dvoch množín realizuje metóda **retainAll()**, kde opäť vieme vložiť nejakú kolekciu. Odstránenie prvku možno dosiahnuť pomocou metódy **remove()**, kde uvedieme neželaný prvok.

```
Set<String> kamaráti = new HashSet<String>();
kamaráti.add("peter@sidlisko.sk“);
kamaráti.add("jana@sidlisko.sk“);

kamaráti.remove("jana@sidlisko.sk“);
// kamaráti.contains("jana@sidlisko.sk“) teraz vráti false
```

Množina vlastných typov

Skúsme reprezentovať **e-maily** pomocou samostatnej triedy **Email:**

```
public class Email {
  private String login;

  private String server;

  public Email(String email) {
    int poziciaZavinaca = email.indexOf("@");

    this.login = email.substring(0, poziciaZavinaca);
    this.server = email.substring(poziciaZavinaca + 1);
  }

  public String getLogin() {
    return login;
  }

  public String getServer() {
    return server;
  }

  public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result
      + ((login == null) ? 0 : login.hashCode());
    result = prime * result
      + ((server == null) ? 0 : server.hashCode());
    return result;
  }
```

```
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Email other = (Email) obj;
    if (login == null) {
      if (other.login != null)
        return false;
    } else if (!login.equals(other.login))
      return false;
    if (server == null) {
      if (other.server != null)
        return false;
    } else if (!server.equals(other.server))
      return false;
    return true;
  }
}
```

Metódy **hashCode()** a **equals()** vygeneroval Eclipse. Množinu odosielateľov potom vieme reprezentovať nasledovne:

```
Set<Email> kamaráti = new HashSet<Email>();
kamaráti.add(new Email("peter@sidlisko.sk“));
kamaráti.add(new Email("jana@sidlisko.sk“));
```

Sumár operácií s množinami:

Operácia	Matematický zápis	Kód
Zjednotenie	$A \cup B$	A.addAll(B)
Prienik	$A \cap B$	A.retainAll(B)
Príslušnosť prvku	$x \in A$	A.contains(x)
Test podmnožiny	$A \subseteq B$	A.containsAll(B)
Vytvorenie jednoprvkovej množiny	$\{ x \}$	Collections.singletonSet(x)

Usporiadané množiny

Zatiaľ sme používali množiny, ktoré nemali definované usporiadanie prvkov. Hoci základná definícia množiny nehovorí nič o usporiadaní, predsa len existuje pojem *čiastočne usporiadanej množiny* (resp. *totálne usporiadanej množiny*), ktorej prvky sú usporiadané na základe istého kritéria. Usporiadanie zaručuje, že pri prechádzaní prvkov množiny ich budeme získavať vždy v rovnakom poradí.

SortedSet a TreeSet

Usporiadanej množine zodpovedá interfejs **java.util.SortedSet**, ktorý implementuje zabudovaná trieda **TreeSet**.

```
SortedSet<String> mená = new TreeSet<String>();
mená.add("Marek“);
mená.add("Ján“);
mená.add("Matúš“);
mená.add("Lukáš“);
```

Ak začneme prechádzať prvkami množiny, zistíme, že ich budeme získavať v abecednom poradí:

```
for(String meno : mená) {
  System.out.println(mená);
}
```

Prirodzené usporiadanie kľúčov

To preto, že **TreeSet** stojí na prirodzenom usporiadaní (*natural ordering*) prvkov, ktoré je v reťazcoch definované implementovaním interfejsu **Comparable** a prekrytím metódy **compareTo()**. (Zmieňovali sme sa o tom v minulých prednáškach).

Ak prvky množiny, ktoré sú tvorené objektami, neimplementujú tento interfejs, alebo ak potrebujeme vlastné kritérium pre usporiadanie objektov, môžeme dodať do konštruktora triedy **TreeSet** vlastnú inštanciu komparátora **Comparator**.

```
Comparator<Film> komparátor = ...
SortedSet<Film> filmy = new TreeSet<String>(komparátor);
```

Vo všetkých ostatných hľadiskách sa usporiadané množiny **SortedSet** správajú ako bežné množiny **Set**, čomu zodpovedá aj hierarchia dedičnosti medzi týmito interfejsmi.

Mapy

V predošlých sekciách sme používali veľké množstvo cudzích pojmov. Set, sorted set, arraylist,... Čo keby sme si chceli spraviť miniaplikáciu, ktorá by predstavovala slovník cudzích slov?

Ako na to? Predovšetkým potrebujeme zvoliť vhodnú reprezentáciu. Každá položka v slovníku by mohla byť reprezentovaná ako dvojica reťazcov: pojem a jeho preklad/význam.

Položka

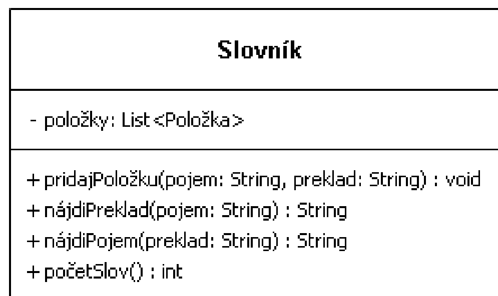
- pojem: String
- preklad: String

```
public class Položka {
  private String pojem;
  private String preklad;

  public Položka(String pojem, String preklad) {
    this.pojem = pojem;
    this.preklad = preklad;
  }

  // .. gettre, setter, hashCode, equals
}
```

Samotný slovník by sme mohli reprezentovať nasledovnou triedou:




```
public class Slovník {

    private List<Položka> položky = new ArrayList<Položka>();

    public void pridajPoložku(String pojem, String preklad) {
          položky.add(new Položka(pojem, preklad));
    }

    public String nájdiPreklad(String pojem) {
          for (Položka p : this.položky) {
                if(p.getPojem().equals(pojem)) {
                      return p.getPreklad();
                }
          }
          System.out.println("Položka sa nenašla.");
          return null;
    }

    public String nájdiPojem(String preklad) {
          for (Položka p : this.položky) {
                if(p.getPreklad().equals(preklad)) {
                      return p.getPojem();
                }
          }
          System.out.println("Položka sa nenašla.");
          return null;
    }
```

```
    public int početSlov() {
        return this.položky.size();
    }
}
```

Položky budeme ukladať do zoznamu **ArrayList**. Vyhľadanie položky podľa pojmu, resp. podľa prekladu získame tak, že prejdeme zoznam a hľadáme prvok s požadovaným pojmom (prekladom) a následne vrátime príslušný preklad (pojem).

0	1	2	3	4
Položka pojem = "class" preklad = "trieda"	**Položka** pojem = "instance" preklad = "inštancia"	**Položka** pojem = "exception" preklad = "výnimka"	**Položka** pojem = "set" preklad = "množina"	**Položka** pojem = "sorted set" preklad = "usporiadaná množina"

Mapa

Situácie, keď v dátovej štruktúre pracujeme s dvojicami prvkov však vieme vyriešiť oveľa elegantnejšie a efektívnejšie. Stačí využiť dátovú štruktúru **mapa** (niekde nazývaná tiež asociatívne pole; slovník; či menej presne hašovacia tabuľka alebo *hash*).

Pojem **slovník** (*dictionary*) asi najpresnejšie vystihuje povahu tejto štruktúry. Nachádzajú sa v nej dvojice kľúč-hodnota, čo zodpovedá slovníkovým pojmom (kľúčom) a ich vysvetleniam (hodnotám). Typicky sa predpokladá, že v slovníku budeme vyhľadávať hodnoty podľa kľúča – presne tak, ako vyhľadávame preklad slova podľa pôvodného znenia. Zároveň sa predpokladá rýchle vyhľadávanie v kľúčoch, ktoré je v bežnom prekladovom slovníku dosiahnuté cez ich abecedné usporiadanie (v implementáciách sa samozrejme používajú odlišné spôsoby, napríklad s využitím hašovacích kódov).

kľúč	class	instance	exception	set	sorted set
hodnota	trieda	inštancia	výnimka	množina	usporiadaná množina

Map a HashMap

Mape zodpovedá interfejs **java.util.Map** s implementáciou **java.util.HashMap**. Vytvorenie inštancie je nasledovné:

```
Map<String, String> slovník = new HashMap<String, String>();
```

Všimnime si dva dátové typy v deklarácii: prvý **String** určuje dátový typ kľúčov a druhý typ (zhodou okolností tiež **String**) dátový typ hodnôt. V našom príklade máme reťazcové kľúče a rovnako reťazcové hodnoty.

Mapa podporuje kľúče a hodnoty ľubovoľného typu. Lomené zátvorky slúžia na vynútenie typovej kontroly, teda aby sme náhodou do nej nevkladali rozličné dátové typy a nemiešali „hrušky s jablkami".

Prázdnosť a veľkosť mapy

Horeuvedeným spôsobom vytvoríme prázdnu mapu, čo si môžeme overiť buď volaním metódy **isEmpty()** alebo testom na veľkosť **size() == 0**.

```
if(slovník.isEmpty())  {
   // mapa je prázdna
}
if(slovník.size() == 0) {
  // mapa je prázdna
}
```

Vkladanie hodnôt

Vkladanie kľúča a hodnoty do mapy je možné realizovať cez metódu **put()**.

```
slovník.put("class”, “trieda”);
slovník.put(“instance”, “inštancia”);
```

Objekty, ktoré predstavujú kľúče v mape, musia implementovať dvojicu metód **hashCode()** a **equals()**. Platia pre ne tie isté zásady, ako pre prvky množiny. V opačnom prípade je správanie mapy nepredvídateľné.

Výber hodnoty

Výber hodnoty z mapy vieme realizovať spôsobom známym zo zoznamov: pomocou metódy **get()**. Na rozdiel od zoznamov však neuvádzame do parametra celočíselný index, ale objekt reprezentujúci kľúč.

```
String prekladTriedy = slovník.get("trieda“)
```

V prípade, že sa požadovanú kľúč v mape nenachádza, metóda vráti **null**.

Prítomnosť hodnoty / kľúča

Niekedy je vhodné vopred zistiť, či mapa obsahuje daný kľúč, resp. hodnotu. Použijeme na to metódy **containsKey()**, resp. **containsValue()**.

```
String pojemInterface = “interface”;
if(slovník.containsKey(pojemInterface) {
  String prekladInterfejsu = slovník.get(pojemInterface);
  System.out.println(prekladInterfejsu);
} else {
  System.out.println(“Slovník neobsahuje preklad.“);
}
```

Jednoznačnosť priradenia

Na konkrétny kľúč však môže byť namapovaná len jedna hodnota. Mapa sa teda správa ako matematická funkcia, kde jednej hodnote z definičného oboru prilieha najviac jedna hodnota. V prípade, že vložíme do mapy novú hodnotu s daným kľúčom, stará hodnota sa prepíše.

```
slovník.put(“instance”, “inštancia”);
System.out.println(slovník.get(“instance”));
// vypíse sa “inštancia”

slovník.put(“instance”, “inštancia triedy”);
System.out.println(slovník.get(“instance”));
// vypíse sa “inštancia triedy”
```

Množina kľúčov

Mapa poskytuje možnosť troch „pohľadov“ na vlastné údaje. Chceme vypísať všetky anglické pojmy, ktoré sú v slovníku uložené? Znamená to, že chceme získať všetky kľúče. *Množina* kľúčov typu **Set** môže byť získaná pomocou metódy **keySet()**:

```
Set<String> kľúče = slovník.keySet();
```

Podľa horeuvedeného obrázka teda získame „prvý riadok“ nazvaný *kľúče*, ktorého prvky budú tvorené množinou {class, instance, exception, set, sorted set}/

Množina hodnôt

Protipólom je množina hodnôt (na obrázku „spodný riadok“), čo v príklade so slovníkom zodpovedá slovenským prekladom pojmov. Získame ho metódou **values()**:

```
Collection<String> preklady = slovník.values();
```

Prvky množiny **preklady** budú tvorené slovami *trieda, inštancia, výnimka, množina, usporiadaná množina*.

Kolekcia hodnôt Collection

Metóda **values()** však vracia inštanciu triedy **java.util.Collection**. To je interfejs, ktorý predstavuje najvšeobecnejšiu skupinu objektov, teda *kolekciu*. Z tohto interfejsu dedia interfejsy pre zoznamy (*lists*), množiny (*sets*) a rady (*queues*) (o frontoch sa zmieňovať nebudeme).

Mapa naozaj vracia len *kolekciu* hodnôt, pretože pokojne sa môže stať, že hodnoty obsahujú duplicitné prvky (čím vypadáva možnosť použitia množiny) a ich poradie je nedefinované (čím vypadáva možnosť použitia zoznamov, kde sú prvky v presne danom poradí určenom indexami). Kolekcia však má všetky tradičné metódy: možno ju iterovať cez **for-each**, možno zisťovať jej veľkosť (**size()**) atď.

Položky mapy

Tretím pohľadom na mapu je pohľad cez jednotlivé položky. Vráťme sa k obrázku zo začiatku kapitoly, kde sme položky reprezentovali triedou **Položka** s dvoma inštančnými premennými pre pojem a preklad, a celý slovník bol zoznam položiek. Tretí pohľad zodpovedá presne tejto filozofii.

Mapu dokážeme potom vnímať ako *množinu položiek (entry set)*, kde každej položke zodpovedá inštancia triedy **java.util.Map.Entry**. Položka **Entry** má presne dve inštančné premenné: kľúč a hodnotu.

Množinu položiek potom vieme získať volaním metódy **entrySet()**.

```
Set<Entry<String, String>> položky = slovník.entrySet();
```

Nezľaknime sa komplikovanej syntaxe s lomenými zátvorkami. Na prvej úrovni deklarujeme, že máme množinu, ktorej elementy sú typu **Entry<String, String>**. V druhom priblížení zistíme, že položka **Entry** má kľúče typu **String** (prvý dátový typ v lomených zátvorkách) a hodnoty typu **String** (druhý dátový typ v lomených zátvorkách).

Následne vieme prechádzať položkami:

```
for(Entry<String, String> položka : položky) {
  String pojem = položka.getKey();
  String preklad = položka.getValue();
}
```

Náš pôvodný príklad so slovníkom vieme potom prepísať nasledovne:

```
import java.util.HashMap;
import java.util.Map;
import java.util.Map.Entry;

public class Slovník {

  private Map<String, String> položky
      = new HashMap<String, String>();

  public void pridajPoložku(String pojem, String preklad) {
    položky.put(pojem, preklad);
  }

  public String nájdiPreklad(String pojem) {
    return položky.get(pojem);
  }

  public String nájdiPojem(String preklad) {
    if (!this.položky.containsValue(preklad)) {
      System.out.println("Položka sa nenašla.");
      return null;
    }
    for (Entry<String, String> položka : this.položky.entrySet()) {
      if (položka.getValue().equals(preklad)) {
        return položka.getKey();
      }
    }
    return null;
  }

  public int početSlov() {
    return this.položky.size();
  }
}
```

Na mapy sa môžeme dívať ako na zovšeobecnenie zoznamov, resp. polí. Zoznam *list* totiž predstavuje mapu z celých čísiel (int) do objektov. Kľúče v zozname však musia narastať, musia začínať nulou a nemôžu byť medzi nimi medzery (nemôže sa teda stať, že v mape máme kľúče 2, 4 – jediná korektná možnosť je mať kľúče 0, 1, 2, 3, 4, kde 0, 2 a 3 prislúcha hodnota **null**).

Z tohto dôvodu sa v niektorých jazykoch nazývajú mapy **asociatívnymi poľami**: mapa je vlastne pole, kde objekt asociujeme s iným objektom.

Usporiadané mapy

Usporiadanie kľúčov v mape možno docieliť zmenou implementácie. Ak využijeme **java.util.TreeMap** (a prípadne interfejs **SortedMap**), získame mapu, ktorá garantuje usporiadanie kľúčov a prechádzanie v pevne danom a jasne definovanom poradí.

SortedMap a TreeMap

SortedMap je usporiadaná mapa (a analógia usporiadanej množiny), teda mapa, ktorej kľúče sú usporiadané podľa prirodzeného usporiadania (*natural ordering*) alebo pomocou dodaného komparátora.

```
SortedMap<Integer, Integer> tabuľkaFaktoriálov =
  new TreeMap<Integer, Integer>();
tabuľkaFaktoriálov.put(0, 1); // 0! = 1
tabuľkaFaktoriálov.put(1, 1); // 1! = 1
tabuľkaFaktoriálov.put(2, 2); // 2! = 2
tabuľkaFaktoriálov.put(3, 6); // 3! = 6
```

```
// kľúče sú usporiadané podľa veľkosti
for(Integer kľúč = tabuľkaFaktoriálov.keySet()) {
  System.out.println(kľúč + "! = " + tabuľkaFaktoriálov.get(kľúč);
}
```

Mapy ako dynamický vzťah medzi dvoma objektami

Mapy je tiež vhodné používať v situáciách, keď chceme počas behu programu ustanoviť vzťah medzi dvoma objektami a nemáme možnosť to urobiť pomocou inštančných premenných.

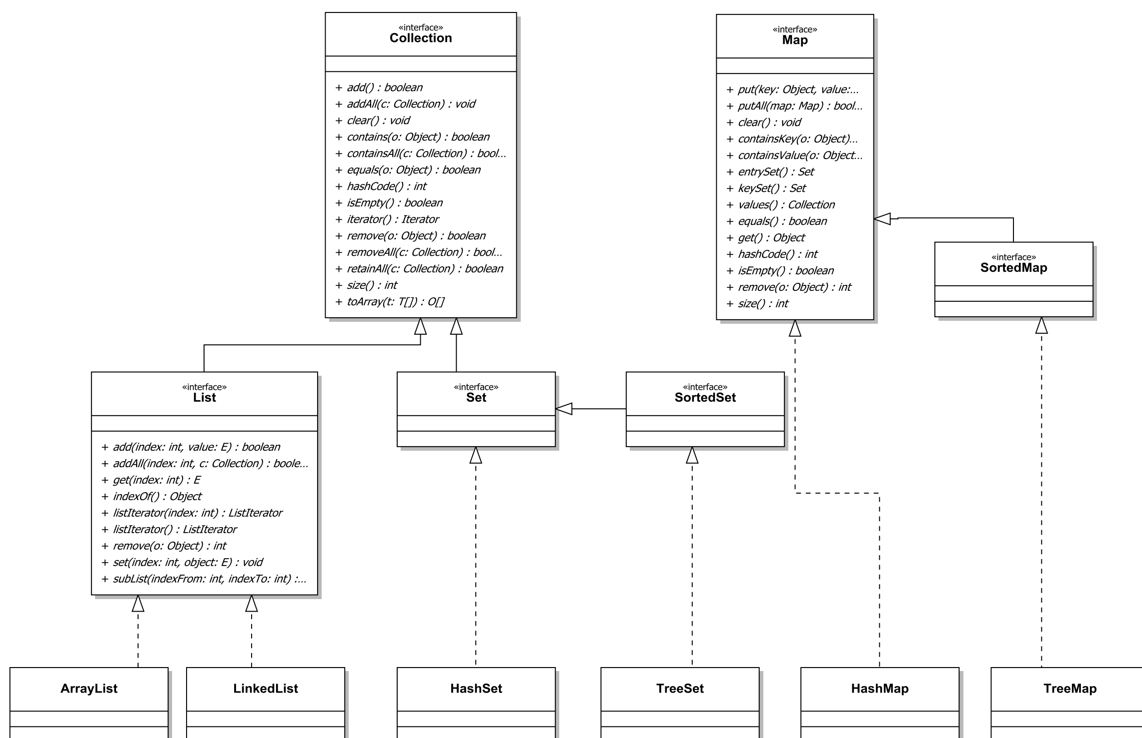
Vezmime si príklad s nákupným košíkom, kde máme triedu **Tovar** s premennými **cena** a **názov**. Ak chceme v nákupnom košíku (trieda **NákupnýKošík**) evidovať, koľko kusov **Tovaru** si zákazník kúpil, napadne nám možnosť dodať do triedy **Tovar** inštančnú premennú **početZakúpenýchKusov**. Čo však v prípade, že triedu meniť nemôžeme? (Mohli by sme oddediť novú triedu **ZakúpenýTovar**, to je však radikálna možnosť).

Alternatívnou možnosťou je vytvoriť v nákupnom košíku mapu z **Tovaru** do čísiel **Integer**, kde budeme evidovať, ktorý tovar si v koľkých kusoch zákazník nakúpil.

```
public class NákupnýKošík {
  private Map<Tovar, Integer> obsah = new HashMap<Tovar, Integer>();

  public void vložTovar(Tovar t) {
    if(obsah.containsKey(t)) {
      int aktuálnyZakúpenýPočet = obsah.get(t);
      obsah.put(t, aktuálnyZakúpenýPočet + 1);
    } else {
      obsah.put(t, 1);
    }
  }
}
```

Vzťahy medzi kolekciami



Vzťahy medzi kolekciami reprezentuje horeuvedený obrázok. Výber môže uľahčiť nasledovná tabuľka:

	Garantované poradie prvkov	Umožnená duplicita prvkov	Indexovaný prístup
Set	-	-	-
Collection	-	áno	-
SortedSet	áno	-	-
List	áno	áno	áno