10.Datové struktury

Tato kapitola popisuje, jak uchovávat více instancí stejného typu (stejné třídy) nebo jeho podtypů. Vzhledem k tomu, že je to poměrně častá úloha, většina programovacích jazyků podporuje seskupování dat stejného typu. V Javě jsou k dispozici tyto základní datové struktury:

- **kolekce** (collections) – se používají pro uchování více prvků stejného typu. Podporují snadné přidávání a ubírání prvků. Kolekce se dělí do následujících tří skupin:
  - **seznamy** (lists) – se nejčastěji používají pro uložení prvků. Prvky se stejným obsahem (hodnotou) mohou být v seznamu vícekrát, seznamy udržují pořadí vkládání prvků. Příkladem může být seznam tiskáren, seznam studentů či seznam věcí v místnosti.
  - **množiny** (sets) – mají proti seznamům dvě odlišnosti: každý prvek lze do seznamu uložit pouze jednou a množiny neudržují pořadí prvků. Výhodou je rychlejší zjišťování, zda prvek již v seznamu je. Množiny lze použít např. pro udržování seznamu typů tiskáren či v nějaké karetní hře pro udržování seznamu karet v ruce hráče.
  - **fronty** (queues) – jsou určeny pro ukládání prvků před dalším zpracováním. Vedle klasických front FIFO Java podporuje i fronty LIFO, fronty s prioritou, fronty se zpožděním či fronty s omezeným přístupem. Fronty nebudou v těchto skriptech dále popisovány.
- **mapy** (maps) – na rozdíl od seznamů se pracuje s dvojicí prvků. První prvek se nazývá klíč a musí být jedinečný, druhý prvek se nazývá hodnota. Výhodou mapy je, že vyhledávání dle klíče je rychlé. Příkladem je frekvenční analýza slov v textu – klíčem bude jednotlivé slovo, hodnotou bude počet výskytů. Dalším příkladem může být seznam bankovních účtů, ke kterým chceme přistupovat rychle pomocí čísla účtu – číslo účtu bude klíč, vlastní účet bude hodnotou. Složitějším příkladem může být přehled známek studentů, kde klíčem bude student a hodnotou je seznam (list) obsahující jednotlivé známky.
- **pole** (array) – představuje seznam s pevným počtem prvků. Podobá se seznamům, ale proti seznamům je však do pole mnohem obtížnější prvky přidávat či ubírat. Pole má i své výhody – je trochu rychlejší a podporuje vkládání primitivních datových typů bez obalování. Používá se v situacích, kdy se nepředpokládá přidávání/ubírání prvků seznamu – seznam parametrů na příkazové řádce, slovo převedené na jednotlivé znaky, seznam nějakých hodnot pro jednotlivé měsíce v roce, atd. Java podporuje i vícerozměrná pole – proto se používá např. pro vyjádření šachovnice či tabulek pevné velikosti.
- **výčtový typ** (enum) – používá se pro uložení pevného seznamů konstantních (neměnných) prvků, každá změna v seznamu znamená nový překlad aplikace. Příkladem může být seznam dnů v týdnu, seznam planet ve sluneční soustavě, seznam barev v kartách. Výčtový typ byl popsán v kapitole 8.

Vedle těchto základních struktur je možné najít v knihovnách Javy další datové struktury, uživatel si může také vytvářet vlastní struktury instancí. V projektu Škola je ukázána možnost, jak vytvořit stromovou datovou strukturu pro uložení hierarchické organizační struktury.

10.1. Collections framework

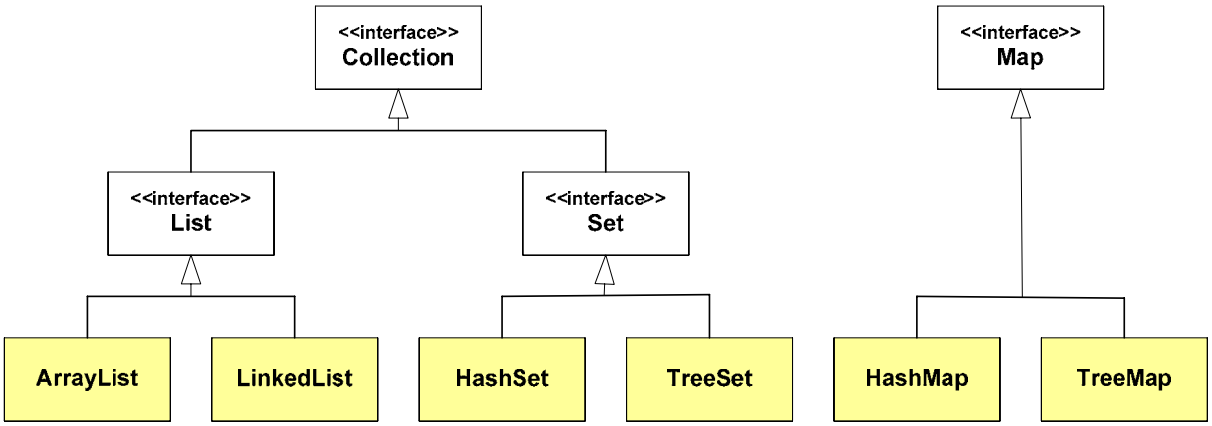
V knihovnách Javy je k dispozici skupina tříd a rozhraní, která je označována jako **Collections Framework**. Jsou součástí standardního API a jsou umístěny v balíčku `java.util`.
Framework se skládá z následujících součástí:

- **rozhraní** – abstraktní datové typy představující jednotlivé druhy kontejnerů. Umožňují manipulovat s kontejnery nezávisle na konkrétní implementaci.
- **konkrétní implementace** – konkrétní implementace rozhraní, připravené k použití.
- **algoritmy** – konkrétní statické metody např. pro setřídění jednotlivých struktur. Tyto metody jsou soustředěny ve třídách **Collections** a **Arrays** (tato třída je určena pro práci s poli).

Ve verzi 5.0 Javy se datové struktury deklarují s využitím tzv. **generických datových typů**, pomocí kterých se určuje typ objektů, které lze vkládat do vytvořených instancí datových struktur. Teorií a deklarací generických datových struktur se v těchto skriptech nebudeme zabývat, v této kapitole si ukážeme používání generických typů v souvislosti s datovými strukturami.

10.2. Základní rozhraní

Následující obrázek obsahuje základní přehled rozhraní pro datové struktury (jsou vynechány fronty) a typické implementace těchto rozhraní.



Obrázek 10.1 Přehled základních rozhraní a konkrétních implementací kolekcí a map

10.3. Kolekce

Rozhraní **Collection** reprezentuje skupinu objektů (instancí) označovaných jako prvky. Jsou zde deklarovány základní operace jako je vložení prvku, zrušení prvku či zjištění počtu prvků. Z variant seskupení prvků budeme popisovat **seznamy** definované rozhraním **List** a **množiny** definované rozhraním **Set**.
V následující tabulce jsou uvedeny nejpoužívanější metody rozhraní `Collection` (tyto metody mají všechny konkrétní implementace rozhraní, tj. seznamy i množiny).

metoda	**užití**
boolean **add**(E element)	Přidá do kolekce prvek typu E. Pokud se operace nepodaří, vrací metoda hodnotu `false`.
void **clear**()	Zruší všechny prvky v kolekci.
boolean **contains**(Object o)	Vrací `true`, jestliže kolekce obsahuje prvek uvedený jako argument metody.
boolean **isEmpty**()	Vrací `true`, jestliže je kolekce prázdná.
Iterator <E>**iterator**()	Vrací instanci rozhraní `Iterator`, pomocí které je možno procházet kolekci
boolean **remove**(Object o)	Jestliže kolekce obsahuje argument, je odpovídající prvek zrušen. Vrací `true`, když je prvek zrušen.
int **size**()	Vrací počet prvků uložených v kolekci.

Tabulka 10.1 Přehled nejpoužívanějších metod rozhraní Collection

10.3.1. Seznamy (List)

Rozhraní **List** reprezentuje seznamy, které mohou obsahovat libovolný počet prvků zadaného typu. Seznamy udržují pořadí vkládaných prvků, umožňují vložit stejný prvek vícekrát. Seznamy umožňují přístup k jednotlivým položkám také pomocí indexů. Tyto indexy jsou typu `int` a jsou v rozsahu `0` až `n-1`, kde `n` je aktuální počet prvků seznamu. Při vkládání lze prvky přidávat na konec nebo vkládat na zadanou pozici – přitom se prvek na této pozici a všechny následující posunou o jedno místo. Prvky lze rušit buď ze zadané pozice, nebo na základě rovnosti se zadaným parametrem (tj. lze zadat instanci, která se má zrušit). Při zrušení prvku se za ním uložené prvky posouvají o jedno místo doleva.
Java nabízí dvě základní implementace seznamů a to třídy **ArrayList** a **LinkedList**. Většinou se používá `ArrayList`, v případě větších změn uvnitř seznamu (vkládání dovnitř seznamu, rušení prvků ze seznamu) je výhodnější používat `LinkedList`.
Všechny implementace rozhraní List mají metody definované pro `Collection` (předchozí tabulka) a navíc metody využívající indexy, uvedené v následující tabulce.

metoda	užití
void **add**(int index, E element)	Přidá do kolekce na zadanou pozici prvek typu E.
E **get**(int index)	Získání prvku na zadané pozici v seznamu.
E **remove**(int index)	Zruší prvek ze zadané pozice seznamu, tento prvek vrací jako návratovou hodnotu.

Tabulka 10.2 Přehled nejpoužívanějších metod rozhraní List

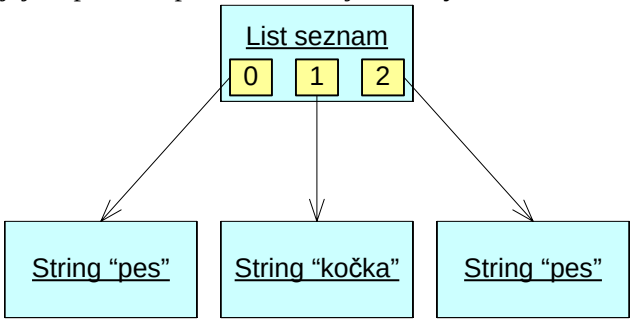
Při vytváření kolekce je třeba určit, jakého typu budou vkládané prvky. Typ prvků se v Javě 5.0 uvádí v ostrých závorkách.
Pro typ prvků je jedno omezení – lze vkládat pouze referenční typy, nelze tedy vytvořit kolekci obsahující prvky typu `int`. Pokud chceme do kolekce ukládat prvky primitivního typu, musíme použít obalový typ např. `Integer`. Při vkládání jednotlivých hodnot a při práci s nimi se využijí automatické konverze mezi primitivními typy a jejich obalovými třídami.
Pokud pro seznam hodnot typu `String` použijeme třídu `ArrayList`, může deklarace a inicializace vypadat takto:

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

Jednotlivé prvky se vkládají pomocí metody `add()`, překladač kontroluje, zda vkládaný prvek je požadovaného typu. Následující kód ukazuje, jak do výše deklarovaného seznamu přidat tři prvky:

```
    seznam.add("pes");
    seznam.add("kočka");
    seznam.add("pes");
```

Prvky jsou v seznamu uloženy v tom pořadí, v jakém se vkládaly. Mají též přiřazeny **indexy,** určující jejich pozici – pozice se číslují od nuly.



Obrázek 10.2 Objekty uložené v seznamu

Ve třídě `ArrayList`, stejně jako v ostatních kontejnerech, je překryta metoda `toString()`. Pro jednoduchý výpis obsahu seznamu je tedy možné použít následující příkaz:

```
System.out.println(seznam);
```

Výsledkem tohoto příkazu je následující výpis:

```
[pes, kočka, pes]
```

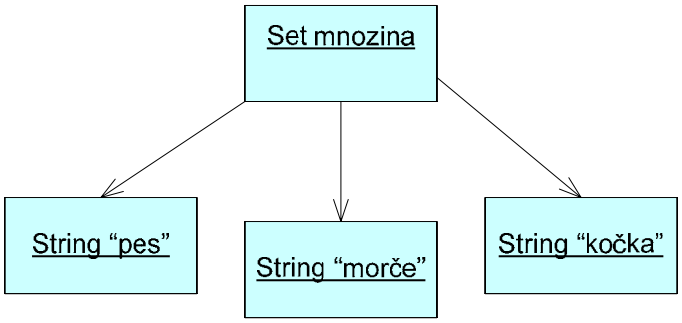
10.3.2. Množiny (Set)

Rozhraní **Set** popisuje datové struktury, které mohou obsahovat libovolný počet prvků zadaného typu, a které (na rozdíl od seznamu) neumožňují vkládat duplicity (vložit dvakrát stejnou instanci či vložit dvě instance, které jsou si rovny). Existují dvě základní implementace, a to **HashSet** a **TreeSet**.
Při ukládání prvků do `HashSet` nelze ovlivnit pořadí prvků – neplatí, že posledně vložený prvek je uložen nakonec. `TreeSet` udržuje své prvky trvale setříděny. Objekty vkládané do instance třídy `TreeSet` musí mít naimplementováno rozhraní **Comparable** (viz dále v této kapitole), neboť metoda `compareTo()` tohoto rozhraní se používá pro zjištění duplicit a při zatřiďování prvků[17].
Rozhraní `Set` nerozšiřuje množství metod, které jsou určeny rozhraním `Collection`.
Při vytváření množin postupujeme obdobně jako u seznamů – při ukládání se navíc kontroluje duplicita vkládaného prvku. Při vložení duplicitní hodnoty nevznikne žádná chyba (nevznikne výjimka), ale ani se duplicitní hodnota nevloží. Pokud se nepodaří objekt vložit, vrátí metoda `add()` hodnotu `false`. Ukážeme si to v následující části kódu:

```
Set<String> mnozina = new HashSet<String>();
mnozina.add("pes");
mnozina.add("kočka");
mnozina.add("pes");
mnozina.add("morče");
System.out.println(mnozina);
```

Výsledkem tohoto kódu je následující výpis:

```
[pes, morče, kočka]
```

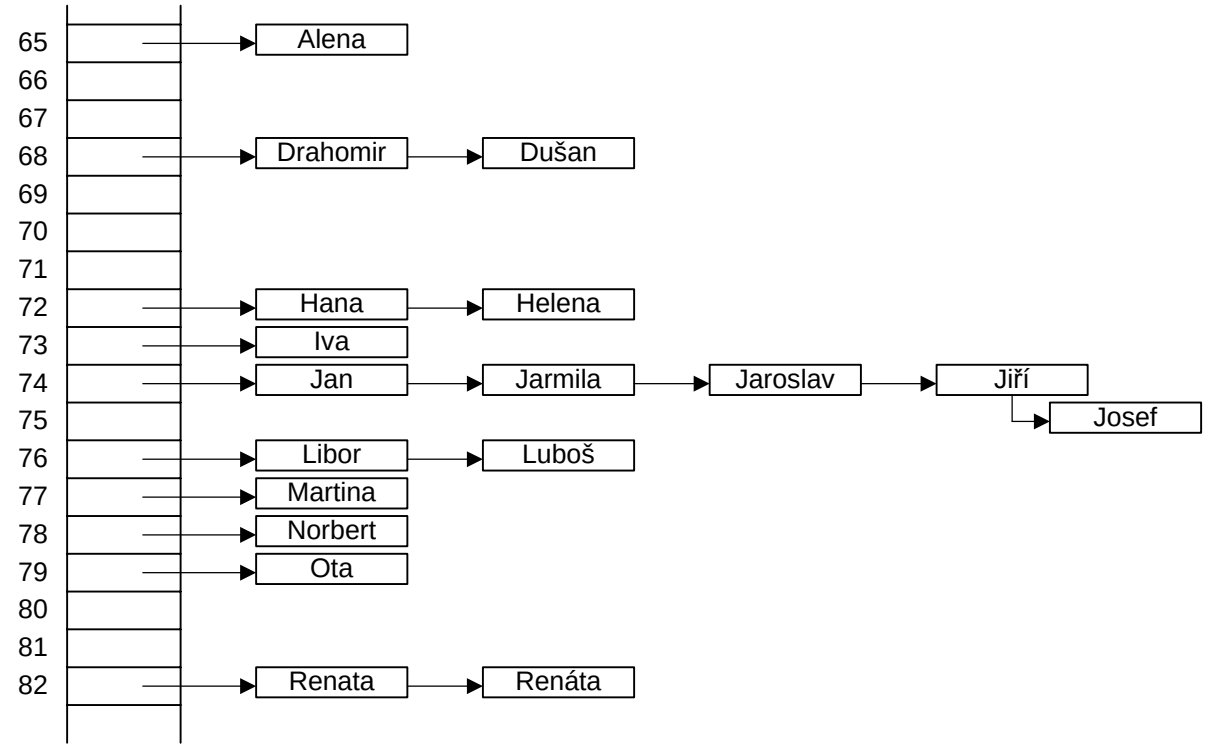


Obrázek 10.3 Objekty uložené v množině (HashSet)

Každá implementace rozhraní `Set` musí zajistit, aby se nevkládaly duplicity. U častěji používané množiny `HashSet` se rovnost objektů zjišťuje pomocí metod `hashCode()` a `equals()`, které jsou popsány v kapitole 6. Pokud u vkládané instance metoda `hashCode()` vrátí číslo odlišné od kódů již vložených instancí, považuje se za jedinečnou. Pokud již v množině existuje instance vracející stejný `hashCode()`, porovnává se s nimi vkládaná instance pomocí metody `equals()`. Následuje

[17] Pokud je v konstruktoru třídy `TreeSet` uvedena implementace rozhraní `Comparator`, tak vkládané prvky nemusí implementovat rozhraní `Comparable`, ale musí být konzistentní s použitou implementací rozhraní `Comparator`.

schématické zobrazení uložení prvků v `HashSet` – pokud prvky vracejí stejný `hashCode()`, uloží se do spojitého seznamu.



Obrázek 10.4 Schématické zobrazení HashSetu – pro stejnou hodnotu hashCode() se prvky ukládají do seznamu

Ze způsobu uložení vyplývají tyto pravidla:

- Jednotlivé objekty by měly vracet pokud možno odlišný `hashCode()`, neboť dle `hashCode()` se vyhledává mnohem rychleji než poté následně pomocí `equals()` ve spojitém seznamu. Z tohoto hlediska není algoritmus pro výpočet `hashCode()` v příkladu na obrázku zvolen příliš vhodně – seznam pro `hashCode()` 74 je poměrně dlouhý.
- Dvě instance, které jsou si rovny dle metody `equals()`, musí vracet stejný `hashCode()` – jinak hrozí, že se uloží duplicitně.
- Metoda `hashCode()` by měla vracet stejnou hodnotu po celou dobu existence instance, obdobně metoda `equals()` by pro porovnání měla používat datové atributy, které se v průběhu existence instance nemění.

Vkládané objekty by měly mít dobře naprogramované metody `hashCode()` a `equals()`.

10.3.3. Procházení kolekce (rozšířený příkaz for)

Pro postupné procházení jednotlivých prvků kolekce se používá speciální syntaxe cyklu `for`, která bývá označována jako „**for each**“. Syntaxe této varianty cyklu `for` je následující:

```
for (Typ identifikátor : kolekce) {
    příkazy;
}
```

První parametr specifikuje typ a identifikátor proměnné, do které budou postupně přiřazovány jednotlivé prvky kolekce, která je uvedena jako druhý parametr. Oddělovačem je v tomto případě dvojtečka. První parametr by měl být toho typu, s jakým je deklarována kolekce.

V následujícím příkladě si ukážeme jak vypsat jednotlivé prvky seznamu na samostatné řádky. Použijeme opět náš seznam zvířat.

```
for (String zvire : seznam){
     System.out.println(zvire);
}
```

Cyklus vypíše následující údaje:

```
pes
kočka
pes
```

Kód na procházení množiny je obdobný:

```
for (String zvire : mnozina){
     System.out.println(zvire);
}
```

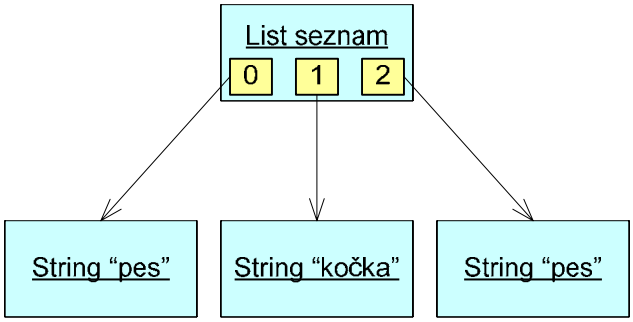
Výsledek bude:

```
pes
kočka
morče
```

Procházení kolekcí pomocí cyklu *for* skrývá jednu záludnost – při procházení (tj. v těle cyklu) nelze mazat prvky kolekce. Pokud je potřeba při procházení kolekce mazat prvky, musí se použít iterátor, který popíšeme v kapitole 10.3.5.

10.3.4. Používání indexů u seznamů

Jak již bylo uvedeno, seznamy (třídy implementující rozhraní List) mají možnost vyžívat **indexy** jednotlivých položek. Indexy začínají od nuly a postupně se zvyšují do velikosti seznamu zmenšeného o jedničku (*seznam.size() – 1*).



Obrázek 10.5 Struktura objektů v seznamu s vyznačením indexů

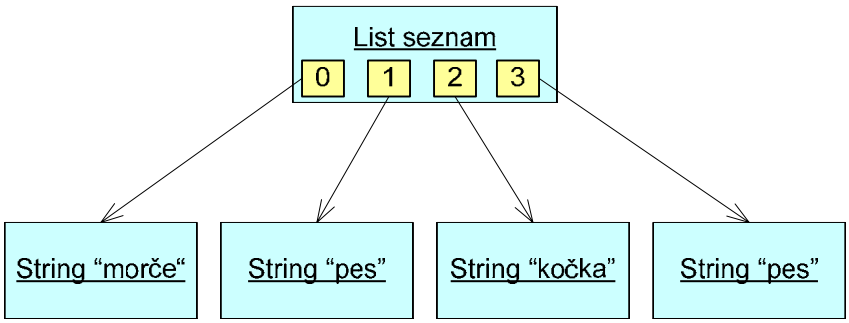
Ze seznamu zobrazeného na obrázku 10.5 získáme druhý prvek seznamu (tj. prvek s indexem 1) příkazem:

```
String zvire = seznam.get(1);
```

Pro vložení prvku do seznamu můžeme použít i variantu metody *add()*, která vloží prvek na zadanou pozici v seznamu. Následující kód ukazuje, jak na první místo v seznamu vložit další prvek.

```
seznam.add(0, "morče");
```

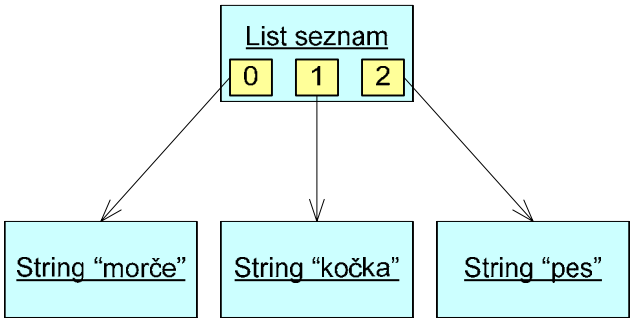
Při vložení prvku na první místo se ostatní prvky posunou – zvýší se jejich indexy o jedničku.



Obrázek 10.6 Struktura objektů v seznamu po přidání morčete

Podobně jako lze vkládat instance na určené místo v seznamu, umožňuje metoda `remove()` rušit prvky na místě určeném konkrétním indexem. Metoda `remove()` vrací rušený prvek jako návratovou hodnotu metody. Následující kód zruší prvek s indexem 1 (rušený prvek dále nepotřebujeme, tudíž vracenou hodnotu nepřiřazujeme do žádné proměnné):

```
     seznam.remove(1);
```



Obrázek 10.7 Struktura objektů v seznamu po smazání prvku s indexem 1

Vzhledem k tomu, že `List` poskytuje přístup k položkám i prostřednictvím indexu, je možno použít také standardní podobu cyklu `for`. Použití tohoto cyklu má smysl v situaci, kdy chceme pracovat i s indexem prvku v seznamu. V následujícím příkladě vypisujeme jednotlivé prvky seznamu na samostatné řádky včetně čísla indexu (používá se formátování pomocí metody `printf()` – viz popis třídy `String`).

```
for (int i = 0; i < seznam.size(); i++) {
     System.out.printf ("%2d. %s",i,seznam.get(i));
}
```

Výstup bude následující:

```
0. morče
1. kočka
2. pes
```

Ani při použití klasického cyklu `for` a indexů nelze při procházení seznamu (uvnitř těla cyklu `for`) mazat prvky seznamu – je potřeba použít iterátor.

10.3.5. Používání iterátoru

V Javě 5.0 se pro procházení kolekcí obvykle používá příkaz `for`(obě jeho varianty). Lze však použít i starší způsob – procházet kolekci pomocí **iterátoru**, instance implementující rozhraní **Iterator**. Proti příkazu `for` má jedinou výhodu – při procházení pomocí iterátoru lze rušit položky v příslušné kolekci.

Iterátor si můžeme představit jako ukazovátko, pomocí kterého se postupně ukazuje na prvky seznamu. Instanci iterátoru získáme pomocí metody `iterator()` příslušné kolekce. Iterátor má tři operace:

metoda	**užití**
boolean **hasNext**()	Vrací true, pokud iterátor ukazuje na prvek v kolekci. Pokud se dojde na konec kolekce, vrací hodnotu `false`.
E **next**()	Vrátí prvek, na který ukazuje iterátor a posune iterátor na další prvek v seznamu.
void **remove**()	V kolekci zruší prvek, který byl naposledy vrácen metodou `next()`. Po `next()` může být volána metoda `remove()` pouze jednou.

Tabulka 10.3 Přehled metod rozhraní Iterator

Při deklaraci iterátoru je potřeba uvést typ prvků v kolekci, která se bude procházet. Deklarace a vlastní procházení může vypadat takto:

```
Iterator<String> iter = seznam.iterator();
while (iter.hasNext()) {
     System.out.println(iter.next());
}
```

Na první pohled je zřetelné, že procházení pomocí cyklu `for` je přehlednější. Iterátor má ve verzi 5.0 jedinou výhodu – umožňuje mazat prvky v průběhu procházení kolekce. Následuje schématický příklad, který ze seznamu vymaže zvířata, která nemám doma.

```
Iterator<String> iter = seznam.iterator();
while (iter.hasNext()) {
     String zvire = iter.next();
     if (! mamDoma(zvire) {
          iter.remove();
     }
}
```

10.3.6. Prohledávání kolekce

Všechny kolekce mají metodu `contains()`, která zjišťuje, zda je v kolekci obsažen aspoň jeden prvek shodný s parametrem. Porovnávání probíhá na základě metody `equals()`, tj. instance vkládané do kolekce musí mít správně implementovánu metodu `equals()`. Mnoho standardních tříd má metodu `equals()` již implementovánu, pro seznam instancí třídy `String` by vyhledávání v kolekci mohlo vypadat takto:

```
if (seznam.contains("morče")) {
     // příkazy
}
```

Pokud potřebujeme rychlé vyhledávání a instance se neopakují, je nejvhodnější použít pro ukládání instancí třídu `HashSet`.
Někdy je vyhledávání složitější – příkladem může být vyhledávání účtu dle čísla účtu. Abychom mohli ukládat účty do `HashSet`, doplníme do třídy `Ucet` metodu `equals()` – dva účty si budou rovny, pokud budou mít stejné číslo účtu:

```
public class Ucet {

     // dosavadní obsah třídy

     public boolean equals (Object o) {
          if (o instanceof Ucet) {
               Ucet druhy = (Ucet)o;
               return cislo == druhy.cislo;
          }
          else {
               return false;
          }
     }

}
```

Nyní v jiné třídě vytvoříme kolekci *seznamUctu*, do které budeme vkládat jednotlivé účty. Tuto třídu rozšíříme i o metodu *existujeUcet()*, která bude vracet *true*, pokud v seznamu účtů existuje účet se zadaným číslem. Pro vyhledání je potřeba vytvořit pomocnou instanci třídy *Ucet*, s jejíž pomocí budeme vyhledávat. Ukázka kódu obsahuje také metodu *getUcet()*, která vrátí účet se zadaným číslem:

```
     private Collection<Ucet> seznamUctu = new HashSet<Ucet>();

     public boolean existujeUcet(int cisloUctu) {
          Ucet pomocny = new Ucet(cisloUctu, "pomocny");
          return seznamUctu.contains(pomocny);
     }

     public Ucet vratUcet(int cisloUctu) {
          for (Ucet ucet : seznamUctu) {
               if (ucet.getCislo() == cisloUctu) {
                    return ucet;
               }
          }
          return null;
     }
```

V těchto situacích bývá výhodnější použít pro uložení místo kolekce mapu – to si ukážeme v podkapitole 10.4.2.

10.3.7. Třídění kolekcí

Častou operací je třídění kolekcí podle různých kritérii. K tomu, abychom mohli třídit, musíme splnit dva předpoklady:

- mít k dispozici metodu, která porovná dva prvky seznamu a zjistí, který prvek je větší a který menší (který má být dříve v seznamu, který později). Existují dvě základní možnosti:
  - tzv. přirozené řazení – třída, jejíž instance se mají třídit, musí implementovat rozhraní **Comparable,**
  - vytvořit speciální třídu implementující rozhraní **Comparator**. Tato třída je „nezávislá“ na třídě, jejíž instance se mají třídit. Toto řešení umožňuje mít více řadících pravidel pro stejné instance (jednou se seznam účtů setřídí dle čísla účtu, podruhé dle hodnoty uložených peněz),
- mít k dispozici třídící algoritmus. Pro jednotlivé typy datových struktur jsou k dispozici různé možnosti:
  - pro setřídění seznamů (implementací **List**) se používá statická metoda *Collections.sort()*,

* pro setřídění polí (**array** – budou vysvětleny na konci kapitoly) se používá `Arrays.sort()`,
* množiny (**Set**) nelze obecně třídit, `TreeSet` jako jedna z implementací udržuje prvky automaticky setříděné,
* mapy (**Map**) též nelze třídit, obdobně jako u množin existuje implementace `TreeMap`, která automaticky udržuje vložené prvky setříděné,
* ve třídě **Collections** jsou též metody `max()` a `min()`, které vyhledají největší/nejmenší hodnotu v kolekci (v seznamech i v množinách),
* obdobně pro pole (**array**) jsou ve třídě **Arrays** metody `max()` a `min()` pro vyhledání největší/nejmenší hodnoty.

Pokud se má použít **přirozené řazení** (natural ordering), musí třída, jejíž instance se mají třídit, implementovat rozhraní **Comparable<T>**, které předepisuje jedinou metodu, a to `public int compareTo (T o)`. V této metodě se porovnává aktuální instance s instancí, předanou jako parametr. Metoda by měla vracet nulu v případě, že jsou si instance rovny (tj. na pořadí těchto dvou instancí v setříděném seznamu nezáleží). Záporné číslo je vráceno v případě, že aktuální instance je „menší“ než instance uvedená jako parametr a kladné číslo se vrací v opačném případě. Výsledky metody `compareTo()` by měly být konzistentní s metodou `equals()` – jsou-li si dvě instance této třídy rovny na základě volání metody `equals()`, měla by metoda `compareTo()` vrátit hodnotu nula. V mnoha standardních třídách je již rozhraní `Comparable` implementováno – instance třídy `String` se řadí vzestupně dle abecedy (anglické), instance obalových tříd se řadí od nejmenšího čísla k největšímu.
My si ukážeme implementaci přirozeného řazení na příkladu s účty – účty se budou řadit vzestupně dle čísla účtu. V deklaraci třídy se musí uvést, že třída implementuje rozhraní `Comparable` a současně typ instancí, se kterými se budou instance této třídy porovnávat (obvykle se porovnává pouze s instancemi stejné třídy). V následujícím kódu vidíte hlavičku třídy a implementaci metody `compareTo()`.

```
public class Ucet implements Comparable<Ucet>{

    private int cisloUctu;
    private String vlastnik;
    private double stav = 0;

    // část kódu není uvedena

    public int compareTo(Ucet druhyUcet){
        if (this.cisloUctu == druhyUcet.cisloUctu){
            return 0;
        }
        else {
            if (this.cisloUctu < druhyUcet.cisloUctu){
                return -1;
            }
            else {
                return 1;
            }
        }
    }

    // další metody třídy Ucet
```

Vlastní setřídění instancí uložených v seznamu (`List`) je nyní již velmi jednoduché – zavolá se statická metoda `sort()` ze třídy `Collections`:

```
    Collections.sort(seznamUctu);
```

Pokud používáme množiny (implementace rozhraní `Set`) a chceme prvky setřídit, musíme použít implementaci **TreeSet**, která automaticky vkládané prvky třídí. Zde platí i obrácené omezení, do `TreeSet` lze ukládat pouze instance, které implementují rozhraní `Comparable` nebo které jsou konzistentní s implementací rozhraní `Comparator` použitou při vytváření instance `TreeSet`. Obdobně je to s ukládáním klíčů do **TreeMap**.
Implementací rozhraní `Comparable` můžeme řadit instance pouze dle jednoho kritéria. Když budeme chtít setřídit účty podle jmen vlastníků (podle abecedy), musíme napsat řazení s využitím rozhraní **Comparator**. Pro třídění instancí jedné třídy lze vytvořit libovolný počet „pomocných tříd“, které implementují rozhraní `Comparator`. Každá z těchto implementací může řadit instance podle jiného kritéria. Nyní si ukážeme, jak může vypadat pomocná třída implementující rozhraní `Comparator`, která bude řadit účty dle vlastníka účtu („podle abecedy“).

```
import java.util.Comparator;

class PorovnavaniUctuDleAbecedy implements Comparator<Ucet> {

     public int compare (Ucet prvni, Ucet druhy){
          String vlastnikPrvni = prvni.getVlastnik();
          String vlastnikDruhy = druhy.getVlastnik();
          return vlastnikPrvni.compareTo(vlastnikDruhy);
     }
}
```

Pro vlastní setřídění seznamu účtů (typu List) zavoláme metodu `sort()` ze třídy `Collections` takto:

```
     Collections.sort(seznamUctu, new PorovnavaniUctuDleAbecedy());
```

Při porovnávání dvou instancí třídy `Ucet` se volá metoda `compare()` uvedená ve třídě `PorovnavaniUctuDleAbecedy`.
Námi vytvořenou implementaci rozhraní `Comparator` je možné použít i při řazení prvků v množině `TreeSet`. Množinu účtu trvale setříděnou abecedně podle vlastníka vytvoříme takto:

```
Set mnozinaUctu = new TreeSet(new PorovnavaniUctuDleAbecedy());
```

Použití tohoto řazení u `TreeSet` má však jeden nepříjemný vedlejší důsledek – do takto definované množiny nelze vložit účty dvou vlastníků stejného jména (či dva účty stejného vlastníka). `TreeSet` patří mezi množiny a již z definice množiny nelze vložit duplicity. U `TreeSet` se při přirozeném řazení rovnost zjišťuje pomocí metody `compareTo()`[18], při použití implementace rozhraní `Comparator` se rovnost zjišťuje pomocí metody `compare()`.
Přirozené řazení i řazení přes implementaci rozhraní `Comparator` je možno využít v dalších metodách třídy `Collections`. Jsou zde např. definovány metody `max()` a `min()`, které vrátí prvek kolekce s největší či nejmenší hodnotou. Pokud je při volání metod `max()` nebo `min()` jako parametr uvedena pouze kolekce, pro vyhledání extrému se použije přirozené řazení pomocí metody `compareTo()`. Vyhledání účtu s nejvyšším číslem účtu je jednoduché:

```
     Ucet ucetSNejvyssimCislem = Collections.max(seznamUctu);
```

Metody `max()` a `min()` jsou přetížené, jako druhý parametr je možno uvést implementaci rozhraní `Comparator`. Pokud budeme chtít vybrat ze seznamu účtů účet s nejvyšší uloženou částkou, potřebujeme následující implementaci rozhraní `Comparator`:

[18] Toto je hlavní důvod, proč by metody `equals()` a `compareTo()` měly být konzistentní – aby při ukládání instancí do množiny byla identifikace duplicit stejná pro implementace `HashSet()` i `TreeSet()`.

```
import java.util.Comparator;

class PorovnavaniUctuDleStavu implements Comparator<Ucet> {

     public int compare (Ucet prvni, Ucet druhy){
          double stavPrvni = prvni.getStav();
          double stavDruhy = druhy.getStav();
          if (stavPrvni == stavDruhy){
               return 0;
          }
          else {
               if (stavPrvni > stavDruhy){
                    return 1;
               }
               else {
                    return -1;
               }
          }
     }
}
```

Spuštění metody pro vyhledání účtu s nejvyšší uloženou částkou bude vypadat následovně:

```
Ucet ucetSNejvyssimStavem = Collections.max(seznamUctu,
                                    new PorovnavaniUctuDleStavu());
```

Pokud existuje více účtů se stejným nejvyšším stavem, vrátí se pouze jeden z nich.

metoda	**užití**
static T **max**(Collection col)	Vrátí maximální (nejvyšší) prvek z kolekce dle přirozeného řazení. Instance v kolekci musí implementovat rozhraní `Comparable`.
static T **max**(Collection col, Comparator comp)	Vrací maximální (nejvyšší) prvek z kolekce v řazení dle komparátoru. Komparátor musí řadit typ instancí uložených v kolekci.
static T **min**(Collection col)	Vrátí minimální (nejmenší) prvek z kolekce dle přirozeného řazení. Instance v kolekci musí implementovat rozhraní `Comparable`.
static T **min**(Collection col, Comparator comp)	Vrátí minimální (nejmenší) prvek z kolekce v řazení dle komparátoru. Komparátor musí řadit typ instancí uložených v kolekci.
static int **frequency**(Collection col, Object o)	Zjišťuje, kolik je v kolekci shodných prvků (dle metody `equals()`) se zadanou instancí.
static void **reverse**(List list)	Obrátí pořadí prvků v seznamu.
static void **shuffle**(List list)	Náhodně zamíchá prvky v zadaném seznamu.
static void **sort**(List list)	Setřídí prvky v seznamu dle přirozeného řazení. Instance v seznamu musí implementovat rozhraní `Comparable`.
static void **sort**(List list, Comparator comp)	Setřídí prvky v seznamu dle zadaného komparátoru. Komparátor musí řadit typ instancí uložených v seznamu.
static void **swap** (List list, int i, int j)	Zamění v seznamu prvky na indexu `i` a na indexu `j`.

metoda	užití
static Comparator **reverseOrder**()	Vrátí komparátor, který bude řadit obráceně vzhledem k přirozenému řazení.
static Comparator **reverseOrder**(Comparator comp)	K zadanému komparátoru vrátí komparátor, který bude třídit v obráceném pořadí.
static Collection **synchronizedCollection**(Collection col)	K zadané kolekci vrátí kolekci, kterou je bezpečné používat ve vláknech. Existují obdobné metody pro implementace `List`, `Set`, `Map` a `SortedMap`.
static Collection **unmodifiableCollection**(Collection col)	K zadané kolekci vrátí kolekci, kterou nelze již upravovat (přidávat/ubírat prvky). Existují obdobné metody pro implementace `List`, `Set`, `Map` a `SortedMap`.

Tabulka 10.4 Metody třídy Collections

10.4. Mapy (Map)

Do mapy se ukládá dvojice objektů: klíč a k němu přiřazená hodnota. Příkladem mapy může být telefonní seznam, ve kterém bude klíčem tel. číslo (jedinečné) a k němu jméno osoby[19]. V aplikaci počítající četnost slov v textu bude klíčem slovo a hodnotou počet výskytů.
Často se mapa používá i v situaci, kdy chceme zajistit rychlý přístup k prvkům seznamu dle klíče – např. můžeme vytvořit mapu, kde klíčem bude číslo účtu a hodnotou bude instance třídy `Ucet`.
Použití map má jedno logické omezení – v mapách nemohou být duplicitní klíče.
Mapy se používají i v jiných jazycích, používá se ale často odlišné označení: asociativní pole, slovník (dictionary), hašovací tabulka (ve verzi 1.0 Javy byla k dispozici mapa pojmenovaná **HashTable**, která je stále k dispozici z důvodu zpětné kompatibility).
Základní funkčnost mapy je definována v rozhraní Map, v následující tabulce jsou popsány základní metody:

metoda	užití
V **get**(Object key)	Vrací hodnotu odpovídající zadanému klíči.
V **put**(K key, V value)	Vloží klíč a hodnotu do mapy. Pokud již klíč v mapě je, přepíše se pouze hodnota.
V **remove**(Object k)	Vrací hodnotu odpovídající zadanému klíči a zároveň v mapě ruší odpovídající záznam.
int **size**()	Vrací počet prvků uložených v mapě.
boolean **isEmpty**()	Vrací `true`, jestliže je mapa prázdná.
void **clear**()	Zruší všechny prvky v mapě.
boolean **containsKey**(Object key)	Pokud je klíč obsažen v mapě, vrací `true`.
boolean **containsValue**(Object value)	Vrací `true`, jestliže mapa obsahuje hodnotu uvedenou jako parametr metody.
Set<K> **keySet**()	Vrací množinu (`Set`) obsahující klíče.

[19] Pokud na jednom tel. čísle bude více osob, tak se buď jejich jména spojí do jednoho řetězce či se použije deklarace mapy, která bude mít jako hodnotu seznam (`List`) obsahující jednotlivé osoby.

metoda	**užití**
Collection<V> **values**()	Vrací kolekci (`Collection`) hodnot.
Set<Map.Entry<V,K>> **entrySet**()	Vrací množinu (`Set`) s prvky `Map.Entry`.

Tabulka 10.5 Přehled nejpoužívanějších metod rozhraní Map

V Javě jsou dvě univerzální implementace mapy – **HashMap** a **TreeMap** – a několik speciálních implementací. `HashMap` je rychlejší, `TreeMap` automaticky třídí klíče.
Při deklaraci mapy a při volání konstruktoru je možné (a velmi vhodné) určit typy klíčů a typy hodnot. Následuje příklad mapy, kde klíčem bude řetězec a hodnotou počet výskytů – při deklaraci čísel musíme použít obalovou třídu.

```
    private Map <String, Long> pocetSlov =
                                    new HashMap<String, Long>();
```

Mapa s telefonním seznamem, kde klíčem je telefonní číslo a hodnotou informace o osobě (instance třídy `Osoba`), může být deklarována a inicializována takto:

```
    private Map <Long, Osoba> telSeznam =
                                    new HashMap<Long, Osoba>();
```

Třída, jejíž instance se vkládají jako klíče, by měla implementovat metody `hashCode()` a `equals()`, platí stejná pravidla jako pro instance vkládané do množiny `HashSet`.
Pro ilustraci používání map si uvedeme jednoduchou mapu, jejímiž prvky budou údaje o domácích zvířatech, jejich druh a počet jedinců tohoto druhu. Deklaraci takové mapy provedeme následovně:

```
    HashMap <String, Integer> mapa = new HashMap<String, Integer>();
```

Mapu naplníme následujícími údaji:

```
    mapa.put("pes", 3);
    mapa.put("pes", 2);
    mapa.put("kočka", 1);
    mapa.put("morče", 1);
```

Stejně jako u kolekcí i u map je možné pro jednoduchý výpis obsahu mapy použít příkaz:

```
    System.out.println(mapa);
```

Výsledkem bude následující výpis:

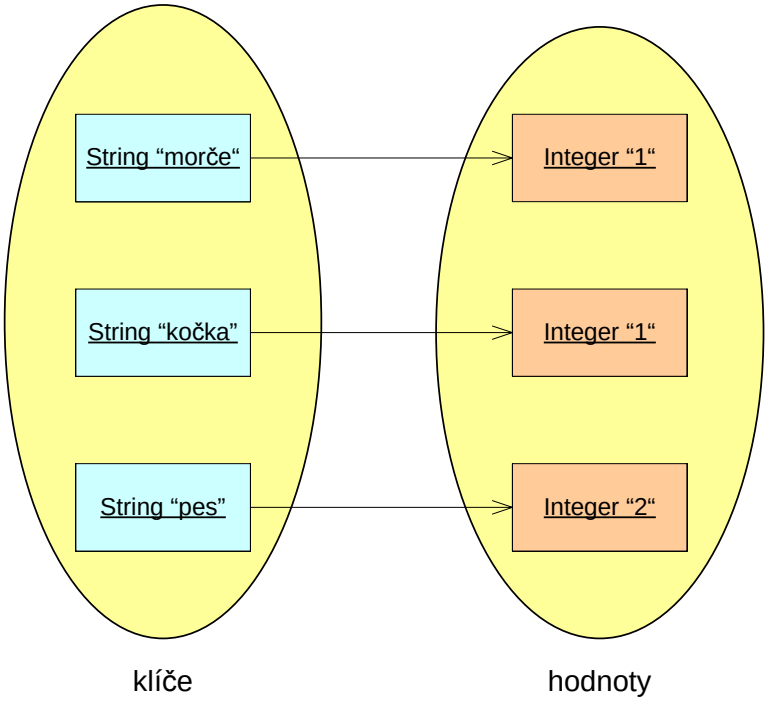
```
    {pes=2, morče=1, kočka=1}
```

Jak si můžete všimnout, při vkládání dvojic se shodnými klíči dochází k přepsání původní hodnoty. V naší mapě to znamená, že původní hodnota 3 ke klíči „pes“ byla přepsána později vkládanou hodnotou 2.
Uvědomte si též, že při vkládání čísel se uplatňuje autoboxing – automatický převod primitivních čísel na obalovou třídu `Integer`. I v následujícím kódu na zvětšení počtu psů o jednoho se několikrát uplatní autoboxing[20]:

```
    mapa.put("pes", mapa.get("pes") + 1);
```

Pro zjišťování jedinečnosti klíče se v `HashMap` využívají metody `hashCode()` a `equals()`.
V `TreeMap` se pro porovnání a zatřídění klíčů používá metoda `compareTo()` – instance vkládané jako klíče musí implementovat rozhraní `Comparable`.
Mapy poskytují pro vyhledávání dvě metody: **containsKey()** pro vyhledávání v klíčích a **containsValue()** pro vyhledávání mezi uloženými hodnotami. Hledání přes klíče je ve většině situací rychlejší.

[20] Díky autoboxingu je tento kód relativně pomalý. Pokud by vkládání do mapy a přičítání hodnot bylo kritickým místem aplikace, lze tyto operace optimalizovat za cenu menší přehlednosti kódu.



Obrázek 10.8 Zobrazení mapy – vztah klíčů a hodnot

10.4.1. Procházení mapy

Mapa se obvykle prochází přes množinu klíčů, ke které získáme přístup pomocí metody *`keySet()`*. Hodnotu ke klíči získáme pomocí metody *`get()`*. Následuje ukázka procházení naší mapy obsahující údaje o zvířatech pomocí cyklu *`for`*:

```
public void vypisMapy(){
     Set<String> mnozinaKlicu = mapa.keySet();
     for (String klic : mnozinaKlicu){
          System.out.println(klic+"\t"+mapa.get(klic));
     }
}
```

Výsledek výpisu metody bude vypadat takto:

```
     pes       3
     morče     1
     kočka     1
```

Mapu lze procházet též pomocí iterátoru – smysl to má však pouze v situaci, kdy chceme při průchodu rušit prvky v mapě. V následujícím kódu zrušíme z mapy všechny druhy, od kterých nemáme doma alespoň jednoho jedince:

```
public void ruseniNeexistujicich(){
     Iterator<String> ukazovatko = mapa.keySet().iterator();
     while(ukazovatko.hasNext()) {
          String klic = ukazovatko.next();
          if (mapa.get(klic) < 1) {
               System.out.println ("ruším z mapy " + klic);
               ukazovatko.remove();
          }
     }
}
```

Mapu lze procházet ještě dvěmi způsoby – přes seznam hodnot, který získáme metodou *values()*, a přes množinu dvojic, kterou vrátí metoda *entrySet()*. Procházení přes seznam hodnot se používá výjimečně, neboť k hodnotě nelze dohledat klíč.
Procházení přes dvojice se též mnoho nepoužívá, neboť je většinou pomalejší, než procházení přes množinu klíčů. Metoda *entrySet()* vytvoří množinu speciálních prvků typu **Map.Entry**, každý prvek vznikne spojením klíče a příslušné hodnoty do jednoho objektu. Prvek typu *Map.Entry* poskytuje metody *getKey()* a *getValue()*, které vracejí jednotlivé části (tj. klíč a jemu příslušnou hodnotu) odpovídajících typů. Následující metoda *vypisMapy3()* ukazuje možné použití těchto metod pro vypsání obsahu mapy.

```
public void vypisMapy3(){
     Set<Map.Entry<String,Integer>> mnozinaDvojic = mapa.entrySet();
     for(Map.Entry <String, Integer> dvojice : mnozinaDvojic) {
          System.out.println(dvojice.getKey()+"\t"
                                                  +dvojice.getValue());
     }
}
```

10.4.2. Mapa pro urychlení vyhledávání

Mapu lze někdy výhodně použít pro urychlení vyhledávání v seznamu – hodnotu, dle které chceme vyhledávat uložíme jako klíč. Ukážeme si to na seznamu účtů, které chceme vyhledávat dle čísla účtu. Klíčem bude číslo účtu (*Integer*), hodnotou bude instance třídy *Ucet*. Při prohledávání mapy dle klíče se využívá automatická konverze typu *int* na obalový typ *Integer*.

```
private Map<Integer,Ucet> seznamUctu = new HashMap<Integer,Ucet>();

public boolean existujeUcet(int cisloUctu) {
     return seznamUctu.containsKey(cisloUctu);
}
public Ucet vratUcet(int cisloUctu) {
     return seznamUctu.get(cisloUctu);
}
```

Použití mapy při vyhledávání má proti kolekcím tyto výhody[21]:

- bývá rychlejší, pokud v instanci je více datových atributů, než jen klíč,
- vlastní kód psaný programátorem je o něco kratší.

Mapa má však i své nevýhody:

- zabírá více místa v paměti (klíče jsou uloženy dvakrát),
- klíč musí být jedinečný.

10.4.3. Mapy obsahující seznamy

Někdy potřebujeme ke klíči vložit do mapy více hodnot – příkladem může být telefonní seznam firmy, kdy k některému tel. číslu chceme vložit více osob. Telefonní čísla máme uvedena v tabulce 10.6.
Pro uložení telefonních čísel použijeme mapu, kde klíčem bude teefonní. číslo a hodnotou bude seznam (*List*) osob, které jsou na tomto telefonním čísle. Deklarace této mapy bude vypadat následovně:

```
Map <Integer, List<String> > telSeznam;
telSeznam = new HashMap<Integer, List<String>> ();
```

[21] Porovnejte si zde uvedený kód s vyhledáváním uvedeným u kolekcí v kapitole 10.3.6.

telefonní číslo	osoby
101	Petra
102	Jan
103	- volné -
104	Tomáš, Prokop
105	Jana, Eva
106	Petr, Antonín, Petra

Tabulka 10.6 Telefonní seznam, který chceme vyjádřit pomocí mapy

Následuje kód metody na přidání další položky do telefonního seznamu – pokud číslo v seznamu ještě neexistuje, tak se vloží. Pokud číslo již v seznamu existuje, přidá se další jméno mezi osoby na tomto telefonním čísle.

```
void pridejPrvek(int telCislo, String jmeno)  {
     if (mapa.containsKey(telCislo)){
          List <String> seznamJmen = mapa.get(telCislo);
          seznamJmen.add(jmeno);
     }
     else {
          List <String> seznamJmen = new ArrayList<String>();
          seznamJmen.add(jmeno);
          mapa.put(telCislo, seznamJmen);
     }
}
```

Metoda *`vypisSeznam()`* vypíše telefonní seznam, ke každému číslu vypíše jednotlivé osoby.

```
public void vypisSeznam (){
     Set<Integer> seznamKlicu = mapa.keySet();
     for (Integer telCislo :seznamKlicu){
          List <String> seznamJmen = mapa.get(telCislo);
          System.out.print(telCislo + "\t\t");
          for(String jmeno : seznamJmen) {
               System.out.print(jmeno +", ");
          }
          System.out.println();
     }
}
```

10.5. Pole (array)

Pole je struktura jednoduchá na používání, do které lze vkládat větší množství hodnot stejného typu. Pole má následující výhody:

- pokud potřebujeme uložit předem daný počet prvků, je pole ze všech datových struktur nejefektivnější,
- pole umožňuje vkládat přímo primitivní datové typy, u ostatních datových struktur se musí převést na objekty,
- je možné vytvářet jednorozměrná i vícerozměrná pole.

Pole má také nevýhody:

- je třeba předem znát počet prvků, které do něj budete ukládat,
- nepodporuje některé složitější způsoby práce s objekty (např. vkládání prvků doprostřed pole či vytváření asociativních polí),

- špatně se v něm vyjadřuje neexistence prvku – pokud máme pole pro 20 čísel a zatím máme vloženo pouze 10 čísel, potřebujeme pomocnou proměnnou pro vyjádření, která políčka pole jsou neobsazena[22],
- pro pole nejsou definovány žádné speciální metody, lze použít pouze metody třídy `Object` a ty se většinou nevyužívají.

10.5.1. Jednorozměrné pole

Jednorozměrné pole si lze představit jako řadu stejných hodnot. Pole má jeden identifikátor (jméno), pro práci s jednotlivými položkami používáme indexy. Následuje příklad pole se jmény dnů v týdnu.

pondělí	úterý	středa	čtvrtek	pátek	sobota	neděle	← hodnota
0	1	2	3	4	5	6	← index

Pole v Javě patří mezi referenční proměnné a podobně jako u objektů se rozlišuje deklarace pole od vytvoření instance pole (inicializace nebo také alokace pole). Pole se pozná podle hranatých závorek [ ], které se uvádějí při deklaraci, inicializaci i při přístupu k prvkům pole. Jednorozměrné pole lze deklarovat oběma následujícími způsoby, na umístění závorek nezáleží:

```
typ [] jmenoPole
typ jmenoPole []
```

Typ určuje položky pole (např. čísla typu `int` nebo řetězce typu `String`) – všechny prvky pole jsou stejného typu. Rozsah pole při deklaraci neuvádíme, deklarací pouze vytváříme identifikátor pole. Následuje několik příkladů deklarace pole:

```
int [] prvocisla;
String [] dnyVTydnu;
String [] args;
```

Pole inicializujeme stejně jako ostatní referenční proměnné pomocí `new,` za kterým se uvede typ prvků pole a počet položek v hranatých závorkách:

```
jmenoPole = new typ [pocetPoložek];
```

Příklady inicializace (vytváření) pole:

```
prvocisla = new int [5];
dnyVTydnu = new String [7];
```

Deklaraci a inicializaci lze spojit dohromady, vytvoření pole pro pět prvočísel zapíšeme takto:

```
int prvocisla [] = new int [5];
```

Na jednotlivé prvky pole se odkazujeme **indexy**, které se číslují od nuly. Z toho vyplývá, že poslední index je o jedničku menší než velikost pole (počet prvků uvedený při inicializaci). Na první položku námi definovaného pole se odkážeme výrazem `prvocisla[0]`, na druhou `prvocisla[1]` a na poslední `prvocisla[4]`. Java striktně kontroluje překročení mezí pole – pokud použijeme index 5 nebo jiný mimo interval 0 až 4, bude ohlášena chyba za běhu programu (výjimka `ArrayIndexOutOfBoundException`).
Po vytvoření jsou v jednotlivých položkách pole jejich inicializační hodnoty, tj. např. pole celých čísel obsahuje ve všech položkách nuly. Lze vytvořit i pole s již naplněnými hodnotami, pro pole `dnyVTydnu` by deklarace a inicializace vypadala takto (všimněte si, že se zde neuvádí `new`):

```
String dnyVTydnu[] = {  "pondělí", "úterý", "středa", "čtvrtek",
                        "pátek", "sobota","neděle"};
```

[22] Můžeme také říct, že pokud je v nějakém políčku hodnota nula, tak políčko není obsazeno. V tomto řešení ale nemůžeme do pole vložit hodnotu nula.

Práce s tímto polem je naprosto stejná jako s ostatními poli vytvořenými pomocí *new*, prvky takto inicializovaného pole nejsou konstanty (tj. lze je měnit). Jediný rozdíl je v tom, že pokud chceme inicializovat pole tímto způsobem, musíme spojit deklaraci a inicializaci do jednoho příkazu.
Každé pole má proměnnou **length** přístupnou pouze pro čtení, ve které je uložen počet prvků pole. Příklady použití najdete v následující podkapitole.

10.5.2. Procházení jednorozměrného pole

K procházení pole je možno využít dva způsoby. První způsob spočívá v použití klasického cyklu **for** a indexů. Použijeme jej v příkladu jednoduché metody pro součet hodnot prvků pole čísel typu *int*. Pro zjištění délky pole, které je předáno jako parametr, použijeme proměnnou *length*.

```
    public static int secti(int [] cisla) {
         int soucet=0;
         for (int i = 0; i < cisla.length ; i++) {
              soucet += cisla[i];
         }
         return soucet;
    }
```

Druhý způsob procházení pole využívá cyklu „**for each**“. Jako příklad nám opět poslouží metoda *secti()*.

```
    public static int secti(int [] cisla) {
         int soucet=0;
         for (int cislo: cisla) {
              soucet += cislo;
         }
         return soucet;
    }
```

10.5.3. Proměnlivý počet parametrů metody

Pokud potřebujete metodě předat větší počet parametrů stejného typu, je vhodné deklarovat příslušný parametr jako pole. Příkladem může být metoda *secti()* z minulé podkapitoly.
Při volání metody můžeme přímo zadat čísla k sečtení do hlavičky metody:

```
    int vysl = secti(new int [] {4, 6, 10, 20});
```

Od verze 5.0 zavádí Java podporu tzv. proměnlivého počtu parametrů metody, který zjednodušuje hlavně zápis při volání metody s větším počtem parametrů stejného typu. Deklarace metody je téměř stejná jako v předchozím případě, pouze místo hranatých závorek se v hlavičce uvedou tři tečky. Parametry se opět předávají jako pole.

```
    public static int secti(int ... cisla) {
         int soucet=0;
         for (int cislo: cisla) {
              soucet += cislo;
         }
         return soucet;
    }
```

Volání metody s proměnlivým počtem parametrů je mnohem jednodušší a přehlednější – prvky pole parametrů uvedeme přímo v kulatých závorkách metody (první řádek následujícího příkladu). Na místě proměnlivého počtu parametrů lze též uvést pole parametrů (třetí řádek příkladu).

```
    int vysl = secti(4, 6, 10, 20);
    int [] cisla = { 5, 10, 20 };
    int vysl2 = secti(cisla);
```

Není obtížné odvodit, že proměnlivý počet parametrů může být uveden v hlavičce pouze jednou a že musí být poslední. Proměnlivý počet parametrů je použit např. v metodě *format()* třídy *String* (viz kapitola 5), jejíž deklarace vypadá takto:

```
public static String format(String format, Object ... args)
```

Deklaraci s proměnlivým počtem parametrů můžeme použít i u metody *main()*:

```
    public static void main (String ... args) {
        // obsah metody
    }
```

10.5.4. Vícerozměrná pole

V Javě lze vytvářet i vícerozměrná pole, počet dvojic hranatých závorek odpovídá počtu rozměrů. Deklarace vypadají takto:

```
    int poleDvojrozmerne [] [];
```

Inicializovat toto pole lze několika způsoby. Na následujícím řádku:

```
    int poleDvojrozmerne [] [] = new int [2] [3];
```

vznikne pole se dvěma řádky a třemi sloupci, položky jsou naplněny nulami. Lze použít i inicializaci výčtem – následuje příklad vytvoření pole se dvěma řádky a třemi sloupci, prvky jsou naplněny zadanými hodnotami:

```
    int poleDvojrozmerne [ ] [ ] = {
        { 1, 2, 3, },
        { 4, 5, 6, },
    }
```

V Javě je možná i postupná inicializace jednotlivých rozměrů pole:

```
    int poleDvojrozmerne [ ] [ ] = new int [2] [ ];
```

U tohoto pole je již inicializován počet řádků, není ale určen počet sloupců. Tento způsob postupné inicializace umožňuje vytvářet poněkud nezvyklá pole, která budou mít v každém řádku jiný počet prvků[23].

```
    poleDvojrozmerne [0] = new int [3];
    poleDvojrozmerne [1] = new int [5];
```

Nyní má pole v prvním řádku tři a ve druhém pět prvků. Všechny prvky mají hodnotu nula.
Pro zjištění počtu prvků lze opět použít proměnnou *length*. Počet řádků našeho dvojrozměrného pole zjistíme výrazem *poleDvojrozmerne.length*, počet prvků prvního řádku *poleDvojrozmerne[0].length* a pro další řádky analogicky.
Lze vytvářet i pole tří a více rozměrná. Při jejich deklaraci a inicializaci platí stejná pravidla jako u dvojrozměrných polí. V případě, že pole vytváříme po částech, nesmíme přeskakovat rozměry, nelze tedy napsat

```
    int poleTroj [] [] [] = new int [5] [ ] [5];
```

Pro procházení vícerozměrných polí můžeme použít obě varianty cyklu *for*. V následujícím kódu si to ukážeme na výpisu obsahu dvourozměrného pole čísel typu *int*.

[23] V Javě se vytvářejí pole polí, ne klasická dvourozměrná pole známá např. z Pascalu.

```
public static void vypisPole(int [][]pole){
     for (int[] radek : pole){
          for (int prvek : radek) {
               System.out.print(prvek + ", ");
          }
          System.out.println();
     }
}

public static void vypisPole2(int[][]pole){
     for (int i = 0; i < pole.length; i++){
          for (int j = 0; j < pole[i].length; j++) {
               System.out.print(pole[i][j] + ", ");
          }
          System.out.println();
     }
}
```

10.5.5. Parametry vstupní řádky

Pokud má třída poskytnout možnost spuštění z příkazové řádky, musí obsahovat metodu `main` deklarovanou s hlavičkou `public static void main (String [] args)`. Parametrem metody je tedy jednorozměrné pole řetězců. Při spuštění programu můžeme za příkaz `java` a jméno třídy uvést libovolný počet parametrů oddělených mezerou, které budou při spuštění metody `main` uloženy do pole, na které odkazujeme identifikátorem `args`. Pole `args` má samozřejmě proměnnou `length`, která udává počet zadaných parametrů. Pokud nejsou zadány žádné parametry, má proměnná `args.length` hodnotu nula. V případě, že některý parametr má obsahovat mezery, je nutné ho zadat na příkazové řádce v uvozovkách, např. "Dobrý den".
Následující příklad vezme z příkazové řádky desetinné číslo představující poloměr kruhu a na konzole vypíše jeho obvod a obsah. Parametr je předán jako `String` – před prováděním operací je tedy nutný převod na číslo. Bližší informace ke konstrukci `try catch` najdete v kapitole 12 věnované výjimkám.

```
 1  public class ObsahKruhu {
 2
 3  public static void main (String [] args) {
 4    double polomer = 0;
 5    if (args.length == 1) {
 6       try {
 7            polomer = Double.valueOf(args[0]).doubleValue();
 8            double obvod = 2*Math.PI*polomer;
 9            double obsah = Math.PI*polomer*polomer;
10            System.out.println("Kruh s poloměrem "+polomer
11                 +" má obsah "+obsah+" a obvod "+ obvod);
12       }
13    catch (NumberFormatException e) {
14            System.out.println("Zadaný parametr není číslo");
15            System.exit(1);
16       }
17    }
18    else {
19       System.out.println("Nebyl zadán poloměr kruhu, nelze" +
       "spočítat obsah ");
20    }
21  }
22  }
```