PAZ

12. prednáška (7.12.2011)

Java Collections Framework

Skoro všetko, čo programátor potrebuje

Java

Spomínate si?

```
Turtle[] noveLopty = new Turtle[this.lopty.length+1];
for (int i=0; i<this.lopty.length; i++)
     noveLopty[i] = this.lopty[i];
noveLopty[noveLopty.length-1] = lopta;
this.lopty = noveLopty;
```

```
public void vlozNoveDvd(Dvd dvd) {
    Dvd[] novePole = new Dvd[filmy.length+1];
    System.arraycopy(filmy,0,novePole,0,filmy.length);
    filmy = novePole;
    filmy[filmy.length-1] = dvd;

}
```

Polia v Jave (nevýhody?)

- **Dĺžka poľa** (počet políčok/prvkov) je určená **pri** jeho **vytvorení** (cez **new:** `new Lopta[5]`)
  - Počet prvkov poľa nemožno zmeniť

- **Finty** na zmenu veľkosti poľa:
  - Vytvoríme nové pole „vhodnej“ veľkosti, vykopírujeme do neho obsah políčok pôvodného poľa a zmeníme referenčnú premennú tak, aby referencovala nové pole
  - Vyrobíme dostatočne veľké pole (napr. 1000 prvkov) a v nejakej *int* premennej si pamätáme koľko políčok “zľava” má „platný obsah“ (“platný” počet prvkov)

ArrayList – „dynamické pole“

- Trieda: **java.util.ArrayList<E>**
  - od verzie Java 1.5

“Generická” trieda

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

Typ políčok `ArrayList`-u

ArrayList – metódy

- `int size()` – vráti dĺžku (počet prvkov)
- `E get(int index)` – vráti obsah políčka na zadanom indexe (číslujeme od 0)
- `void set(int index, E hodnota)` – nastaví obsah políčka na zadanom indexe
- `boolean add(E hodnota)` – na koniec „zoznamu“ pridá zadanú hodnotu (počet prvkov sa zvýši o 1)
- `void clear()` – vyprázdni zoznam (počet prvkov bude 0)

ArrayList – príklad

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

slova.add("Dobry"); slova.add("den");

System.out.println(slova.toString());

slova.set(1, "vecer");

for (int i=0; i<slova.size(); i++)
     System.out.println(slova.get(i));
```

Pridávame na koniec zoznamu (ArrayList-u)

Meníme obsah uložený na indexe 1

ArrayList vs. polia

`String[] slova;`	`ArrayList<String> slova;`
`slova = new String[10];`	`slova = new ArrayList<String>();`
`slova[2] = "Ahoj";`	`slova.set(2, "Ahoj");`
`String s = slova[1];`	`String s = slova.get(1);`
`slova.length`	`slova.size()`

`ArrayList` má navyše: `add`, `remove`, `clear`, `toString`, `indexOf`, ...

Veľkosť `ArrayList`-u sa mení len cez metódy `add` a `remove`!!

ArrayList nie je dokonalý?

```
ArrayList<Turtle> korytnacky =
    new ArrayList<Turtle>();
ArrayList<FilmNaDVD> filmy =
    new ArrayList<FilmNaDVD>();
```

```
ArrayList<int> cisla =
    new ArrayList<int>();
```

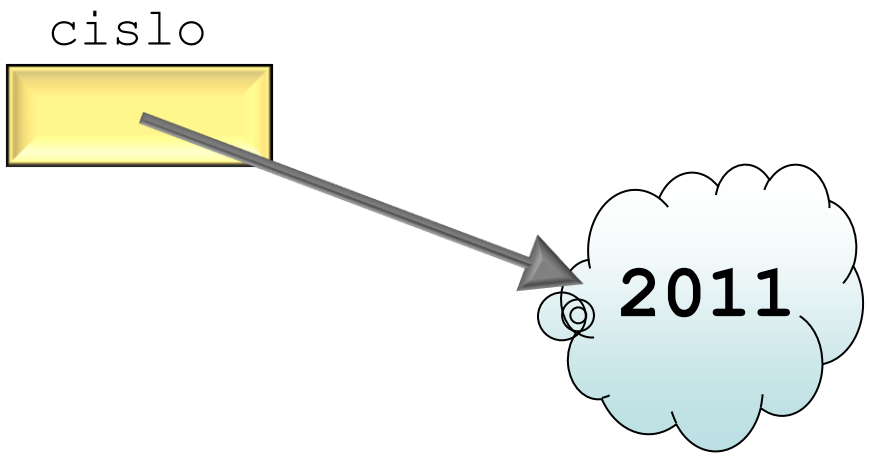
`ArrayList<int>` nebude fungovať!

Medzi < > môžeme dať **len triedu**, nie primitívny typ. `ArrayList` uchováva referencie na objekty zadanej triedy ...

Zabaľme hodnoty do objektov

- Java ponúka **obaľovacie** (wrapovacie) triedy na zabalenie hodnôt primitívnych typov do objektov:
  - **Integer** - **`int`**
  - **Long** - **`long`**
  - **Byte** - **`byte`**
  - **Short** - **`short`**
  - **Character** - **`char`**
  - **Boolean** - **`boolean`**

```
Integer cislo =
    new Integer(2011);
```



Balíme a vybaľujeme ...

```
Integer cislo = new Integer(2011);
Integer cislo2 = new Integer(2012);
```

Získanie „zabalenej“ hodnoty

```
int c = cislo.intValue();

if (cislo.intValue() < cislo2.intValue()) { }
if (cislo.equals(cislo2)) { }

Character znak = new Character('a');
char z = znak.charValue();
```

Autoboxing ("tajné" skratky)

- **Autoboxing** - riešenie pre lenivých ...

```
Integer cislo = new Integer(2009);
```

```
Integer cislo = 2009;
```

Automatické "zabalenie" do objektu triedy `Integer`

Automatické získanie (rozbalenie) "zabalenej" hodnoty

```
int c = cislo;
```

```
int c = cislo.intValue();
```

ArrayList čísel a autoboxing

```
ArrayList<Integer> cisla =
    new ArrayList<Integer>();
```

Skratka pre:
`cisla.add(new Integer(10));`

```
cisla.add(10);
int prveCislo = cisla.get(0);
```

Skratka pre:
`int prveCislo = cisla.get(0).intValue();`

Algoritmy s ArrayListom

- Vytvorme triedu **Zoznamar**, ktorej objekty budú poskytovať pár užitočných metód na spracovanie zoznamov (*ArrayList*-ov) čísel:
  - `int sucet(ArrayList<Integer> zoznam)`
    - spočíta súčet všetkých čísel v zozname
    - `null` hodnoty vynecháva (pozor, **políčka** zoznamu **sú referencie** na objekty triedy `Integer` a teda **môžu byť** aj `null`)
  - `boolean lenKladne(ArrayList<Integer> zoznam)`
    - `true` práve vtedy, ak všetky čísla v zozname sú kladné

ArrayList nie je jediný!

- Okrem triedy `ArrayList<E>` existuje v Jave aj trieda LinkedList<E>, ktorá robí presne to isté ...
- Načo sú 2 triedy, ktoré robia to isté?
  - Keď dvaja robia to isté, nie je to to isté ...
- **ArrayList**-y skrývajú v sebe pole, ktoré sa pri `add` a `remove` metódach zväčšuje/zmenšuje na základe „finty“ s vytvorením nového poľa
  - v skutočnosti je tam **o dosť chytrejšia** implementácia, než sme používali my (s kapacitou)

ArrayList vs. LinkedList

- O(1) = „rýchlo“, O(n) = „pomaly“ (v najhoršom prípade)

	ArrayList<E>	LinkedList<E>
`get`	O(1)	O(n)
`add` na začiatok alebo koniec	O(n)	O(1)
`remove` prvého alebo posledného	O(n)	O(1)
`set`	O(1)	O(n)

- Každý z nich je **výhodnejší pre iné** praktické situácie
  - pri častom vkladaní a odoberaní z konca je lepší `LinkedList`

ArrayList + LinkedList = List

- Obe triedy majú spoločné to, že **reprezentujú nejaký zoznam** objektov a majú príslušné „užitočné“ metódy pre zoznamy
- Rozhranie **List<E>** je „zoznam“ tých metód, ktoré by mal každý slušný „zoznamový“ objekt implementovať

- `ArrayList<E>` **aj** `LinkedList<E>` implementujú rozhranie `List<E>`!
  - Upravme triedu `Zoznamar` …

Pripomeňme si interface

- Interface = pomenovaný **zoznam hlavičiek** metód
  - hlavička metódy = názov, návratový typ, zoznam typov parametrov

```
public interface Rozhranie { ... }

public class Trieda implements Rozhranie { ... }
```

Trieda prehlasuje, že bude mať všetky metódy, ktoré sú uvedené v rozhraní.

```
Rozhranie o = …;
```

Premenná `o` je schopná referencovať objekt ľubovoľnej triedy, ktorá prehlásila, že implementuje interface `Rozhranie`

Rozhranie `List<E>`

- Predpisuje **základné metódy** na prácu so zoznamami:
  - *add, remove, get, set, clear, size, isEmpty*
- Metódy na **prácu s väčším počtom** prvkov:
  - *addAll, removeAll*
  - *subList* - vráti zoznam reprezentujúci podzoznam prvkov
  - *toArray* - na základe zoznamu vyrobí klasické Java pole a naplní ho podľa zoznamu
- http://download.oracle.com/javase/6/docs/api/java/util/List.html

Rozhrania v praxi

```
public int sucet(ArrayList<Integer> zoznam)
```

Kde sa len dá, používame namiesto triedy (implementácie) rozhranie.

```
public int sucet(List<Integer> zoznam)
```

Množina (`Set<E>`)

- **Množina** je skupina objektov
  - žiadne poradie
  - každý prvok sa tam nachádza len raz
- Rozhranie: **`Set<E>`**

- Implementácie:
  - triedy implementujúce rozhranie `Set<E>`
  - **`HashSet<E>`**, `TreeSet<E>`, `LinkedHashSet<E>`
    - každá z tried robí rôzne veci rôzne efektívne

Metódy rozhrania `Set<E>`

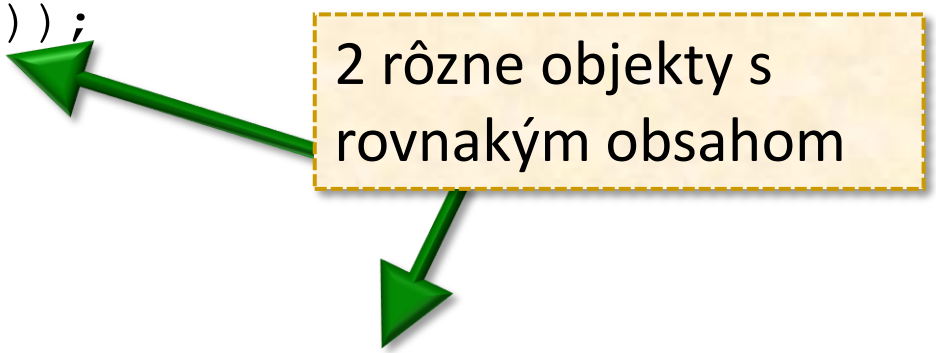
- `int size()` – vráti počet prvkov množiny
- `boolean contains(E o)` – vráti, či objekt *o* je v množine
- `boolean add(E o)` – pridá do množiny objekt *o*
- `boolean remove(E o)` – odstráni z množiny objekt *o*
- `void clear()` – vyprázdni množinu

- `boolean` návratová hodnota hovorí, či operácia spôsobila zmenu obsahu množiny
  - Ak do množiny {1, 2, 3} pridáme číslo 2, množina sa nezmení ...
  - Ak z množiny {1, 2, 3} odoberieme číslo 5, množina sa nezmení

Množinové záhady?

- Vypíše sa **`true`** alebo **`false`**? Prečo?

```
Set<Integer> cisla = new HashSet<Integer>();
cisla.add(new Integer(2011));
```




```
System.out.println(cisla.contains(new Integer(2011)));
```

- Ako overiť, či všetky prvky v množine majú nejakú vlastnosť? Ako sa k nim dostať?
  - máme `size()`, ale nemáme `get(int index)` ako pri zoznamoch `List<E>`

Záhada č. 1: Zhodnosť objektov

- Aj **rôzne objekty**, môžu reprezentovať (uchovávať) **rovnaký obsah**:
  - Príklady: `String, Integer, Double, ...`

```
String retazec1 = new String("Ahoj");
String retazec2 = new String("Ahoj");
retazec1.equals(retazec2);
```

Na zistenie toho, či obsah objektov je rovnaký sa používa metóda `equals` ...

Rodokmeň metódy `equals` (1)

- Metóda `equals` je definovaná v triede `Object`
  - Dôsledok: Každý Java objekt má metódu `equals`
- `equals` overuje, či iný objekt má rovnaký obsah ako objekt, nad ktorým túto metódu voláme

```
String retazec1 = new String("Ahoj");
Turtle k = new Turtle();
if (retazec1.equals(k)) { … }
```

Reprezentujú objekty referencované z `retazec1` a `k` ten istý obsah? Určite nie.

Rodokmeň metódy `equals` (2)

- Ako je implementovaný `equals` v `Object`-e?

```
public boolean equals(Object o) {
    return this == o;
}
```

Test: Ten istý objekt?

- Niektoré triedy prekryli metódu `equals` („preprogramovali“ správanie), tak aby fungovala zmysluplnejšie (napr. `String`, `Integer`, …)
  - Zvyknime si: objekty testujeme na rovnosť cez metódu `equals` (spoľahnime sa na tvorcu triedy)

Testujeme metódu equals

- Vytvorme triedu `Bod`, ktorá bude uchovávať x-ovú a y-ovú súradnicu nejakého bodu:
  - to, aké súradnice má bod reprezentovaný objektom, je určené pri volaní konštruktora (žiadne `setX` a `setY`)
  - náš `Bod` bude podobný `String`-u: po vytvorení objektu jeho obsah nemožno zmeniť
- Preprogramujme metódu `equals` ...
  - dva body sú rovnaké, keď reprezentujú bod s tými istými súradnicami

Equals pre Bod

```
public boolean equals(Object obj) {
    if (obj == null)
        return false;

    if (obj == this)
        return true;

    if (!(obj instanceof Bod))
        return false;

    Bod objBod = (Bod)obj;
    return (objBod.x == this.x) &&
           (objBod.y == this.y);
}
```

Ja nie som rovnaký ako `null`!

Som rovnaký ako ja sám!

Môžem sa porovnávať iba s iným `Bod`-om!
(upozornenie: vzhľadom na dedičnosť nie je tento test úplne v poriadku)

Pretypujem referenciu...

Porovnám sa s `objBod` (resp. s `obj`) po „zložkách“

Equals len v tandeme ...

- Aj keď sme prekryli `equals` pre triedu `Bod`, práca s množinou a bodmi nefunguje ...
- Dôvod: `equals` funguje v tandeme s metódou `hashCode`
  - metóda `hashCode` definovaná v triede `Object`
- Vždy musí platiť (implikácia):

  Ak majú dva objekty rovnaký obsah (`equals` vráti **`true`**), potom volania metódy `hashCode` vrátia rovnakú hodnotu.

Čo s hashCode?

```
public int hashCode() {
    return 0;
}
```

Splníme podmienku z predchádzajúceho slajdu.
Pozor: takto sa to v praxi nerobí!!!

- Pamätajme:

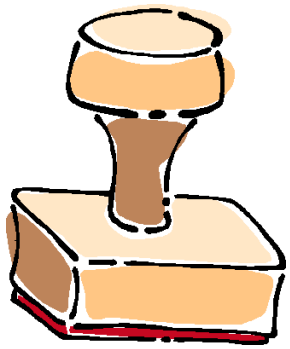
  Ak `o1.equals(o2)`,

  potom `o1.hashCode() == o2.hashCode()`

- `Set<Bod>` už konečne beží ...
- O vygenerovanie metód `equals` a `hashCode` radšej požiadajme Eclipse

Čo je to hashCode?

- `hashCode` je funkcia, ktorá vráti **odtlačok obsahu** objektu
  - rovnaký obsah znamená rovnaký `hashCode` (odtlačok)



- Prečo je `hashCode` užitočný?
  - porovnanie cez `equals` môže byť výpočtovo náročné
  - ak si pamätám `hashCode` obsahu, viem dať veľmi často rýchlu zápornú odpoveď bez vykonania `equals`
    - Ak sú `hashCode`-y rôzne, tak objekty sú rôzne ...
  - `hashCode` pre `String` (“naivný“, nie reálny):
    - súčet kódov znakov modulo 1000

Záhada č. 2: Prvky množiny ...

- Ako zistiť, že všetky prvky množiny majú nejakú vlastnosť, keď k nim nemáme prístup?
- Riešenie:
  - **for-each cyklus**
  - **iterátory**
- Poznámka: riešenia fungujú pre všetky triedy implementujúce rozhranie `Iterable<E>`
  - `Iterable<E>` je zdedené v rozhraniach `Set<E>`, `List<E>`, …

Iterátor

```
Iterator<Bod> it = body.iterator();

while (it.hasNext()) {
    Bod b = it.next();
    System.out.println(b);
}
```

Povieme množine, aby vytvorila iterátor `Bod`-ov

Pracujeme podobne ako so `Scanner`-om

- Cez metódu `remove` iterátora vieme bezpečne odstrániť prvok naposledy vrátený cez `next` z tej kolekcie (množina, zoznam, ...), ktorú práve iterujeme.

For-each cyklus

Premenná, do ktorej postupne ukladáme prechádzané prvky

Cez prvky čoho prechádzame

```
for(Bod b: body)
    System.out.println(b);
```

- For-each cyklus funguje aj pre polia ...

Iterácia – prechod prvkami

- **Kolekcia** – spoločné pomenovanie pre niečo, čo obsahuje nejako nejaké prvky (množina, zoznam, atď.)
- Počas iterácie prvkami kolekcie **nikdy nemodifikujeme** (nepridávame, neodoberáme, nemeníme) iterovanú kolekciu!!! (inak je zle)
  - Povolené je len volanie metódy `remove` iterátora
- Finta:
  - Prvky, ktoré chceme odstrániť, uložíme „bokom“ do novej kolekcie (napr. množina) a po skončení iterácie ich (po jednom alebo naraz) odstránime.

Ukážka bezpečného odstránenia

```
public void odstranParne(Set<Integer> cisla) {
    Set<Integer> naOdstranenie =
        new HashSet<Integer>();

    for(Integer cislo: cisla)
        if (cislo % 2 == 0)
            naOdstranenie.add(cislo);

    for(Integer cislo: naOdstranenie)
        cisla.remove(cislo);
}
```

Vyberieme tie čísla, ktoré chceme odstrániť

Alternatíva:
`cisla.removeAll(naOdstranenie);`

Implementácie `Set<E>`

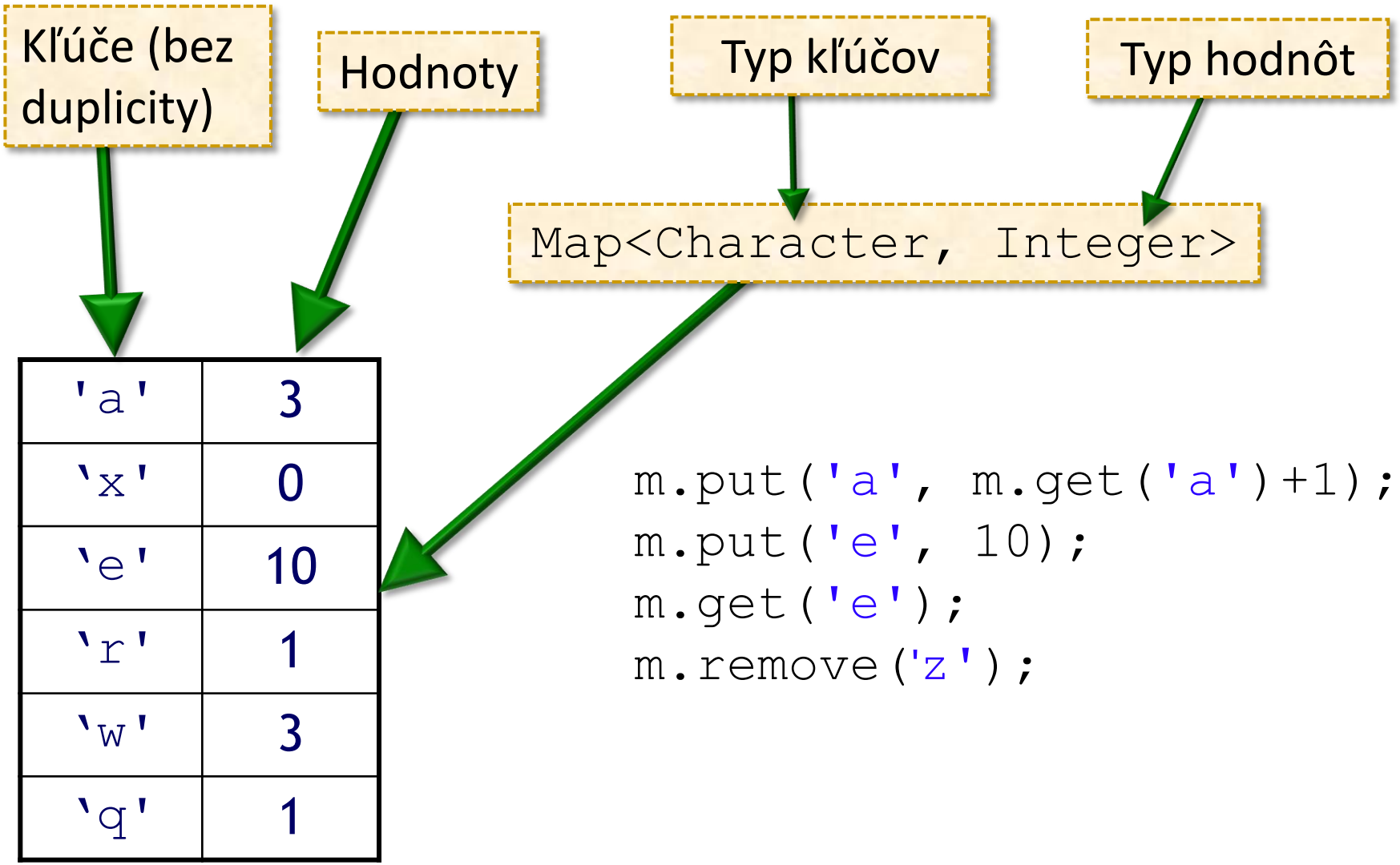
- **HashSet<E>**
  - najčastejšia voľba, založená na tzv. hašovaní
  - prvky sa iterujú v bližšie neurčenom poradí
- **TreeSet<E>**
  - uchováva prvky množiny usporiadané
    - trieda E musí implementovať rozhranie `Comparable<E>`

      alebo
    - musí byť zadaný `Comparator<E>`, ktorý vie porovnávať prvky
  - prvky sa iterujú v poradí od najmenšieho

Map<K, V>

- **Mapa** je mapovaním **kľúčov** typu K na **hodnoty** typu V
  - Každý kľúč môže mať nanajvýš jednu hodnotu
  - Žiadna duplicita kľúčov
- **Metafory:**
  - Zobrazenie z množiny kľúčov do množiny hodnôt
  - Asociatívne pole: hodnoty políčok nie sú prístupné cez indexy, ale cez kľúče ľubovoľného typu
  - Množina párov: <Kľúč, Hodnota>
  - Tabuľka s 2 stĺpcami
    - Frekvenčná tabuľka slov je `Map<String, Integer>`

Map ako tabuľka




```
m.put('a', m.get('a')+1);
m.put('e', 10);
m.get('e');
m.remove('z');
```

Metódy rozhrania `Map<K, V>`

- `int size()` - vráti počet kľúčov (párov)
- `V put(K key, V value)` – nastaví novú hodnotu mapovanú ku kľúču *key* a vráti predchádzajúcu hodnotu
- `V get(K key)` – vráti hodnotu mapovanú k danému kľúču
- `V remove(K key)` – odstráni hodnotu kľúča a aj kľúč samotný z `Map`-u, vráti pôvodne uloženú hodnotu
- `void clear()` - vyprázdni `Map`

Metódy rozhrania Map<K, V>

- `boolean containsKey(K key)` - vráti či `Map` obsahuje zadaný kľúč
- `boolean containsValue(V value)` - vráti či `Map` obsahuje zadanú hodnotu pri nejakom kľúči
- `Set<K> keySet()` - vráti množinu kľúčov, ktoré majú v `Map`-e definovanú hodnotu
- `Set<Map.Entry<K,V>> entrySet()` - vráti množinu položiek (párov) uložených v `Map`-e
  - vrátené `Set`-y sú prepojené s `Map`-om (napr. odstránenie prvku z množiny kľúčov vymaže záznam v `Map`-e)

Map v praxi

```
Map<String, Integer> m = new HashMap<String, Integer>();
m.put("ahoj", 1);
m.put("ahoj", m.get("ahoj") + 1);
m.put("java", 1000);

for (String kluc: m.keySet())
    System.out.println(kluc + ": " + m.get(kluc));
m.remove("java");

// pre pokročilých
for (Entry<String, Integer> par: m.entrySet())
     par.setValue(par.getValue() + 1);
```

Implementácie `Map<K, V>`

- **HashMap<K, V>**
  - interne využíva `HashSet` na uloženie kľúčov
  - najčastejšia voľba
- **TreeMap<K, V>**
  - interne využíva `TreeSet`
  - kľúče sú vždy usporiadané (*K* musí implementovať `Comparable` alebo sa musí definovať `Comparator<K>`)
- **LinkedHashMap<K, V>**
  - interne využíva `LinkedHashSet`
  - kľúče sú v poradí vloženia do `Map`-u

Java Collections Framework

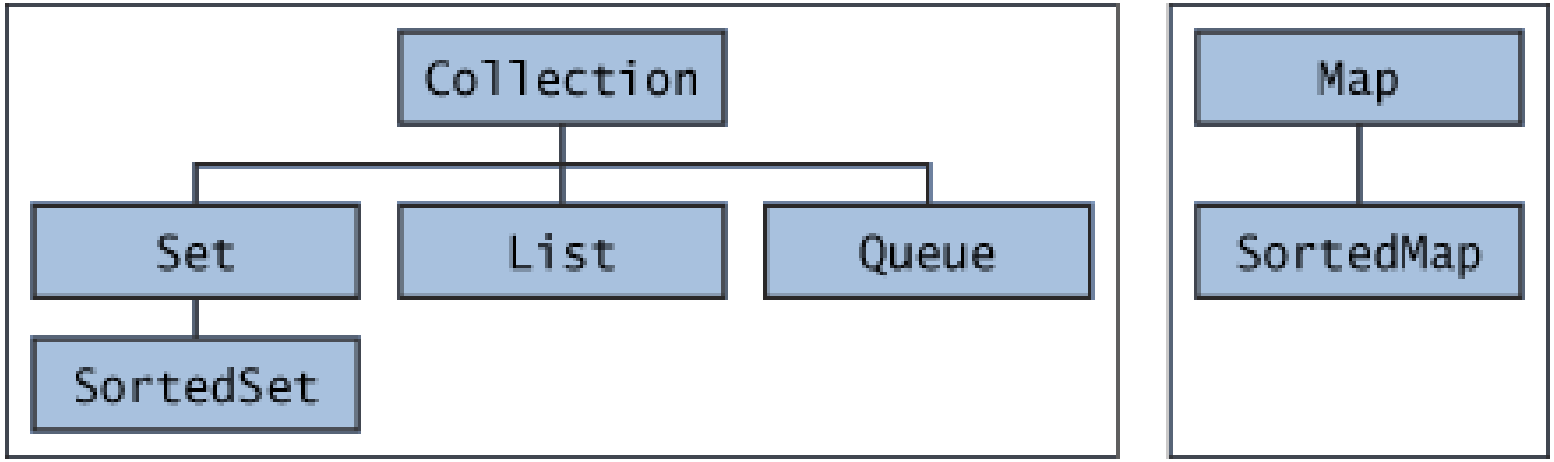
- Má 3 základné zložky:
  - **rozhrania** (`List, Set, Map, …`)
  - **implementácie** rozhraní (`ArrayList, HashSet, LinkedHashSet, …`)
  - **algoritmy** (dostupné cez `Collections.`)

rozhrania

implementácie

algoritmy

Rozhrania JCF



- **`Collection<E>`**
  - najvšeobecnejšia skupina (kolekcia) prvkov typu E
  - rozširujú ju rozhrania `Set<E>`, `List<E>` a `Queue<E>`
- SortedSet/SortedMap - usporiadané prvky/kľúče

Prehľad rozhraní a ich implemenácií

Rozhrania	Implementácie (triedy)				
	Hašovacia tabuľka	Pole	Strom	Spájaný zoznam	Hašovacia tabuľka + spájaný zoznam
Set	HashSet		TreeSet		LinkedHashSet
List		ArrayList		LinkedList	
Map	HashMap		TreeMap		LinkedHashMap

Algoritmy

- Tak ako matematické veci sú v `Math` (`Math.abs`, `Math.min`, `Math.sin`, …), algoritmy pre kolekcie sú v **Collections** a pre polia v **Arrays**.

- Príklady:

```
List<Integer> cisla = …;
Collections.sort(cisla);
Collections.reverse(cisla);
int minimum = Collections.min(cisla);
int pocetVyskytov2 =
  Collections.frequency(cisla, 2);
```

Take-home message ...

- Skoro **všetko**, čo bežný programátor potrebuje, je **už** v Jave (chytro) **naprogramované**
- Princípy fungovania JCF a jeho ponuku
  - pamätajme na `equals`, `hashCode` a autoboxing
- Detaily jednotlivých rozhraní, metód, tried a algoritmov si netreba pamätať (ale treba mať prehľad), **všetko je v dokumentácii** ku API
  - http://java.sun.com/javase/6/docs/api/java/util/package-summary.html
  - http://java.sun.com/docs/books/tutorial/collections/index.html

ak nie sú otázky...

Ďakujem za pozornosť !