

Uspořádané kolekce

```
<link rel="stylesheet" href="http://cdnjs.cloudflare.com/ajax/libs/font-awesome/3.1.0/css/font-awesome.min.css">
```

Úvod

Motivace

- chceme prvky v kolekci uspořádané, ale nechceme to dělat "ručně"
- v kolekci typu `String` chceme jména od K po M

Implementace

- uspořádání dané třídy musí být definováno ve třídě

Rozhraní `Comparable<T>`

- rozhraní slouží k definování **přirozeného** (defaultního) **uspořádání** třídy
- třída implementuje rozhraní \Rightarrow objekty jsou vzájemně *uspořadatelné*
- použití zejména u uspořádaných kontejnerů
- předepisuje jedinou metodu `int compareTo(T o)`
- `T` = typ objektu, název třídy



[Javadoc třídy `Comparable<T>`](#)

Metoda `compareTo`

```
int compareTo(T that)
// used as e1.compareTo(e2)
```

- metoda porovná 2 objekty — `this` (`e1`) a `that` (`e2`)
- vrací celé číslo, pro které platí:
 - číslo je záporné, když $e1 < e2$
 - číslo je kladné, když $e1 > e2$
 - 0, když nezáleží na pořadí
- na samotném čísle nezáleží, je v pořádku používat pouze hodnoty -1, 0, 1

Implementace `Comparable<E>`

```
public class Point implements Comparable<Point> {
    private int x;
    // ascending order
    public int compareTo(Point that) {
        return this.x - that.x;
    }
}
...
new Point(1).compareTo(new Point(4)); // -3
```



Existuje i beztypové rozhraní `Comparable`, to ale nebudeme používat!

`compareTo` vs. `equals`

- chování `compareTo` by mělo být konzistentní s `equals`
- pro rovné objekty by `compareTo` mělo vrátit `0`
- není to však nutnost
 - např. třída `BigDecimal` pro přesné hodnoty podmínku porušuje
 - pro stejné hodnoty s rozdílnou přesností — např. `4.0` a `4.00`
- `compareTo` na rozdíl od `equals` nemusí vstupní objekt přetypovávat a může vyházovat výjimku

Více uspořádání

Co kdybychom chtěli více typů uspořádání, nebo alternativu k přirozenému uspořádání?

Nemůžeme nadefinovat stejnou metodu víckrát.

- rozhraní `Comparator<T>` slouží k definování uspořádání zvnějšku — pomocí objektu jiné třídy
- předepisuje jedinou metodu `int compare(T o1, T o2)`
- uspořádání funguje nad objekty typu `T`
- návratová hodnota `compare` funguje stejně jako u `compareTo`
- funguje jako alternativa pro další uspořádání

Příklad komparátoru

Třída `String` má definované přirozené uspořádání lexikograficky.

Definujme lexikografický komparátor, který ignoruje velikost písmen:

```

public class IgnoreCaseComparator implements Comparator<String> {
    public int compare(String o1, String o2) {
        return o1.toLowerCase().compareTo(o2.toLowerCase());
    }
}
...
new IgnoreCaseComparator().compare("HI", "hi"); // 0

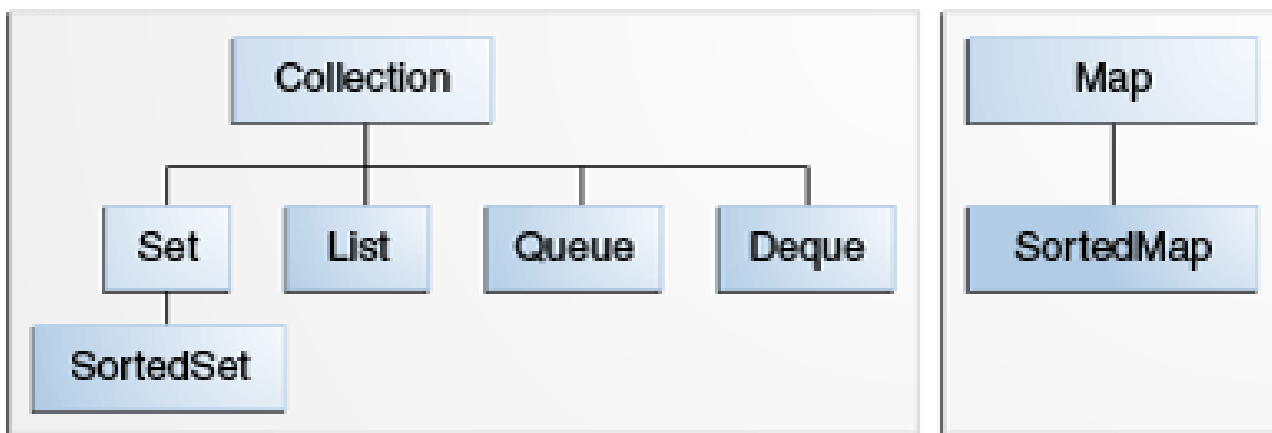
```

Skutečné použití

- metody pro uspořádání programátor v kódu obvykle nepoužívá
- namísto toho používá **uspořádané kolekce**
- prvky v kolekcích jsou řazeny automaticky
- je nutno definovat přirozené uspořádání nebo použít komparátor, aby kolekce věděla, podle jakých pravidel prvky seřadit

Hierarchie rozhraní kolekcí

Budeme se zabývat rozhraními *SortedSet* a *SortedMap*.



SortedSet, SortedMap

SortedSet

- rozhraní pro uspořádané množiny
- všechny vkládané prvky musí implementovat rozhraní *Comparable* (nebo použít komparátor)
- implementace *TreeSet*

SortedMap

- rozhraní pro uspořádané mapy
- všechny vkládané **klíče** musí implementovat rozhraní *Comparable* (nebo použít komparátor)

- implementace `TreeMap`

Konstruktory `TreeSet`

- `TreeSet()`
 - vytvoří prázdnou množinu
 - prvky jsou uspořádány podle **přirozeného uspořádání**
- `TreeSet(Collection<? extends E> c)`
 - vytvoří množinu s prvky kolekce `c`
 - prvky jsou uspořádány podle **přirozeného uspořádání**
- `TreeSet(Comparator<? super E> comparator)`
 - vytvoří prázdnou množinu
 - prvky jsou uspořádány podle **komparátoru**
- `TreeSet(SortedSet<E> s)`
 - vytvoří množinu s prvky i uspořádáním podle `s`

Příklad `TreeSet` I

Definice přirozeného uspořádání:

```
public class Point implements Comparable<Point> {
    ...
    public int compareTo(Point that) {
        return this.x - that.x;
    }
}
```

Příklad `TreeSet` II

Použití:

```
SortedSet<Point> set = new TreeSet<>();
set.add(new Point(3));
set.add(new Point(3));
set.add(new Point(-1));
set.add(new Point(0));
System.out.println(set);
// prints -1, 0, 3
```

Jiný příklad TreeSet

Třída `String` má definované přirozené uspořádání lexikograficky.

```
SortedSet<String> set = new TreeSet<>();
set.add("Bobik");
set.add("ALIK");
set.add("Alík");
System.out.println(set); // [ALIK, Alík, Bobik]

SortedSet<String> set2 = new TreeSet<>(new IgnoreCaseComparator());
set2.addAll(set);
System.out.println(set2); // [ALIK, Bobik]
```



`TreeSet` pro porovnávání prvků používá `compareTo` / `compare`, proto má druhá množina pouze 2 prvky!

TreeSet pod lupou

- implementována jako červeno-černý vyvážený vyhledávací strom
 - ⇒ operace `add`, `remove`, `contains` jsou v $O(\log n)$
- hodnoty jsou uspořádané
 - prvky jsou procházeny v přesně definovaném pořadí



[Javadoc třídy TreeSet](#)

TreeMap

- množina klíčů je de facto `TreeSet`
- hodnoty nejsou uspořádány
- uspořádání lze ovlivnit stejně jako u uspořádané množiny
- implementace stromu a složitost operací je stejná



[Javadoc třídy TreeMap](#)

Příklad TreeMap

Klíče jsou unikátní a uspořádané, hodnoty nikoliv.

```
SortedMap<String, Integer> population = new TreeMap<>();
population.put("Brno", -1);
population.put("Brno", 500_000);
population.put("Bratislava", 500_000);

System.out.println(population);
// {Bratislava=500000, Brno=500000}
```

Repl.it demo k uspořádaným množinám a mapám

- <https://repl.it/@tpitner/PB162-Java-Lecture-08-sorted-set-and-map>