

Seznam, množina, iterátory

Table of Contents

Potomci <code>Collection</code>	1
Seznam <code>List</code>	2
Implementace seznamu <code>ArrayList</code>	2
Implementace seznamu <code>LinkedList</code>	2
<code>ArrayList</code> vs. <code>LinkedList</code>	3
Výkonnostní porovnání seznamů	3
Konstruktory seznamů	3
Vytváření kolekcí přes metody <code>of</code>	4
Vytváření kolekcí přes <code>new</code>	4
Na zamyšlení	4
Metody rozhraní <code>List I</code>	5
Metody rozhraní <code>List II</code>	5
Příklad použití seznamu	5
Množina, <code>Set</code>	6
<code>equals</code> & <code>hashCode</code> — opakování	6
Implementace množiny — <code>HashSet</code>	6
<code>HashSet</code> pod lupou	7
Speciální množina <code>LinkedHashSet</code>	7
Další lineární struktury	7
Starší typy kontejnerů	7
Kontejnery a primitivní typy	8
Procházení kolekcí	8
For-each cyklus I	8
For-each cyklus II	8
Iterátory	9
Příklad s <code>while</code>	9
Metody iterátorů	9
Iterátor — příklad	10
Replit.com demo k iterátorům (a komparátorům z příští přednášky)	10
Kontejnery kontejnerů	10
Replit.com demo ke kontejnerům kontejnerů	10

Potomci `Collection`

Podívejme se na potomky rozhraní `Collection`, konkrétně:

Rozhraní `List`

má implementace

- `ArrayList` - seznam na bázi pole, rychlý při přímém přístupu, nejběžnější
- `LinkedList` - spojovaný seznam, méně používaný

Rozhraní `Set`

má implementace

- `HashSet` - množina na bázi hašovací tabulky, nejčastěji používaná

Seznam `List`

- něco jako dynamické pole
- každý uložený prvek má svou pozici — **číselný index**
- index je celočíselný, nezáporný, typu `int`
- umožňuje procházení seznamu dopředu i zpětně - indexem či iterátorem
- lze pracovat i s *podseznamy*, něco jako řezy (slices) v Pythonu:

```
List<E> subList(int fromIndex, int toIndex)
```

Implementace seznamu `ArrayList`

- nejpoužívanější implementace seznamu
- využívá **pole** pro uchování prvků
- při zvětšování/zmenšování se vytváří nové pole a prvky se musejí přesouvat
- rychlý přístup k prvkům dle indexu
- pomalé operace přidávání a odebírání prvků blíže k začátku seznamu (pole, v němž je seznam, se musí realokovat)



[Javadoc třídy `ArrayList`](#)

Implementace seznamu `LinkedList`

- druhá nejpoužívanější implementace seznamu
- využívá **zřetězený seznam** pro uchování prvků
- pomalejší operace přístupu k prvkům dle indexu "uvnitř" seznamu
- rychlejší operace přidávání a odebírání prvků na začátku a na konci, resp. blízko nich

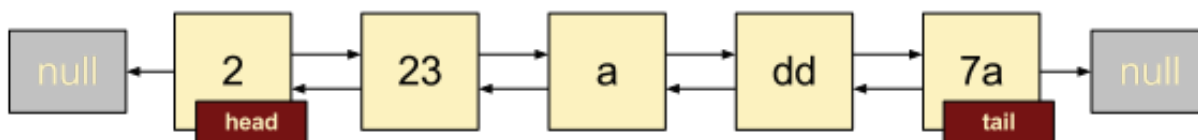


[Javadoc třídy `LinkedList`](#)

ArrayList vs. LinkedList

Array vs. Linked List

Linked List



Array



Kontejnery ukládají pouze jeden typ, v tomto případě `String`.

Výkonnostní porovnání seznamů

List implementation:	ArrayList	LinkedList
add 100000 elements	12 ms	8 ms
remove all elements from last to first	5 ms	9 ms
add 100000 elements at 0th position	1025 ms	18 ms
remove all elements from 0th position	1014 ms	10 ms
add 100000 elements at random position	483 ms	34504 ms
remove all elements from random position	462 ms	36867 ms



Nevhodné případy užití tedy jsou `ArrayList` pro přidávání/odebírání z kraje seznamu a pro dlouhé seznamy `LinkedList` jakékoli přístupy *uprostřed*.

Konstruktory seznamů

```
new ArrayList<>()
```

vytvoří prázdný seznam (s kapacitou 10 prvků)

```
new ArrayList<>(int initialCapacity)
```

vytvoří prázdný seznam s danou kapacitou

`new ArrayList<>(Collection<? extends E> c)`

vytvoří seznam a naplní ho prvky kolekce `c`



Kapacita reprezentuje interní kapacitu, **neznamená** to počet `null` prvků v nové kolekci!

Vytváření kolekcí přes metody `of`

Kromě `new` a konstruktoru lze přes statické tovární metody, kolekce jsou ale pak neměnné:

`List.of(elem1, elem2, ...)`

- vytvoří seznam a naplní ho danými prvky
- vrátí **nemodifikovatelnou** kolekci

`Set.of, Map.of`

analogicky

Vytváření kolekcí přes `new`

Chceme-li kolekci modifikovatelnou, musíme vytvořit novou `new`:

```
List<String> modifiableList = new ArrayList<>(List.of("Y", "N"));
```



Jelikož v množině nejsou duplicity, `Set.of` si hlídá, abychom prvek nepřidali vícekrát: proto selže například `Set.of("foo", "bar", "baz", "foo");`

Na zamyšlení

Jak udělám ze seznamu typu `List` kolekci `Collection`?

- v podstatě nedělám nic - seznam už je kolekci

```
// change the type, it is its superclass  
Collection<Long> collection = list;
```

Jak udělám z kolekce `Collection` seznam `List`?

- musíme vytvořit nový seznam a prvky tam zkopírovat

```
// create new list  
List<Long> l = new ArrayList<>(collection);
```

Metody rozhraní List I

Rozhraní `List` dědí od `Collection`.

Kromě metod v `Collection` obsahuje další metody:

`E get(int index)`

- vrátí prvek na daném indexu
- `IndexOutOfBoundsException` je-li mimo rozsah

`E set(int index, E element)`

- nahradí prvek s indexem `index` prvkem `element`
- vrátí předešlý prvek

Metody rozhraní List II

`void add(int index, E element)`

- přidá prvek na daný index (prvky za ním posune)

`E remove(int index)`

- odstraní prvek na daném indexu (prvky za ním posune)
- vrátí odstraněný prvek

`int indexOf(Object o)`

- vrátí index **prvního** výskytu `o`
- jestli kolekce prvek neobsahuje, vrátí -1

`int lastIndexOf(Object o)`

totéž, ale vrátí index **posledního** výskytu

Příklad použití seznamu

```
List<String> list = new ArrayList<>();
list.add("A");
list.add("C");
list.add(1, "B");
// ["A", "B", "C"]

list.get(2); // "C"
list.set(1, "D"); // "B"
list.indexOf("D"); // 1
```

Množina, Set

- odpovídá matematické představě množiny
- prvek lze do množiny vložit nejvýš *jedenkrát*
- při porovnávání rozhoduje rovnost **podle výsledku volání `equals`**
- umožňuje rychlé dotazování na přítomnost prvku
- provádí rychle atomické operace - se složitostí $O(1)$, *nejhůře* $O(\log(n))$:
 - vkládání prvku — `add`
 - odebírání prvku — `remove`
 - dotaz na přítomnost prvku — `contains`



Množiny jsou primárně bez pořadí, bez uspořádání, existuje však i množina s uspořádáním.

`equals` & `hashCode` — opakování

`equals`

zjistí, jestli jsou objekty obsahově stejné (porovnání atributů).

`hashCode`

vrací pro obsahově stejné objekty stejné číslo, **haš**.

Co je haš?

jakési *falešné ID* — pro různé objekty může `hashCode` vracet stejný haš.

Implementace množiny — `HashSet`

- Ukládá objekty do hašovací tabulky podle haše
- Ideálně konstantní operace (tj. sub-logaritmická složitost)
- Když má více prvků stejný haš, existuje více způsobů řešení
- Pro (ne úplně ideální) `hashCode` $x + y$ vypadá tabulka následovně:

haš	objekt
0	[0,0]
1	[1,0] [0,1]
2	[1,1] [0,2] [2,0]
3	[2,1] [1,2] [0,3] [3,0]



Javadoc třídy `HashSet`

HashSet pod lupou

`boolean contains(Object o)`

- vypočte haš tázaného prvku `o`
- v tabulce najde objekt uložený pod stejným hašem
- objekt porovná s `o` pomocí `equals`

Co když mají všechny objekty stejný haš?

- Množinové operace budou velmi, velmi pomalé.
- Chtěná složitost $O(1)$, $O(\log n)$ zdegeneruje na lineární $O(n)$.

Co když mají stejné objekty různé haše?

- Porušíme kontrakt (předpis) metody `hashCode`.
- Množina `HashSet` přestane fungovat, bude obsahovat duplicity nebo vložený prvek už nenajdeme!

Speciální množina `LinkedHashSet`

- Další implementací množiny je `LinkedHashSet` = `HashSet` + `LinkedList`.
- Zachová pořadí prvků dle jejich vkládání, což jinak u `HashSet` neplatí.

Další lineární struktury

Zásobník

třída `Stack`, struktura LIFO

Fronta

třída `Queue`, struktura FIFO

- fronta může být také *prioritní* — `PriorityQueue`

Oboustranná fronta

třída `Deque` (čteme "deck")

- slučuje vlastnosti zásobníku a fronty
- nabízí operace příslušné oběma typům - vkládání i odběr z obou stran

Starší typy kontejnerů

- Existují tyto starší typy kontejnerů (za → uvádíme náhradu):
 - `Hashtable` → `HashMap`, `HashSet` (podle účelu)
 - `Vector` → `List`
 - `Stack` → `List` nebo lépe `Queue` či `Deque`

Kontejnery a primitivní typy

- Kontejnery ukládají pouze odkazy na objekty, **neukládají primitivní typy**.
- Proto používáme jejich objektové protějšky — `Integer`, `Char`, `Boolean`, `Double`...
- Java automaticky dělá *zabalení*, tzv. **autoboxing**—konverzi primitivního typu na objekt "wrapper".
- Pro zpětnou konverzi se analogicky dělá tzv. **unboxing**, *vybalení*.

```
List<Integer> list = new ArrayList<>();
list.add(new Integer(1));
list.add(1); // autoboxing
int primitiveType = list.get(0); // unboxing
```

Procházení kolekcí

Základní typy:

For-each cyklus

- jednoduché, intuitivní
- nepoužitelné pro modifikace samotné kolekce

Iterátory

- náročnější, ale flexibilnější
- modifikace povolena, např. mazání prvku pod iterátorem

Lambda výrazy s `forEach`

- například `list.forEach(System.out::println)`

For-each cyklus I

- Je rozšířenou syntaxí cyklu `for`.
- Umožňuje procházení kolekcí i **polí**.

```
List<Integer> numbers = List.of(1, 1, 2, 3, 5);
for(Integer i: list) {
    System.out.println(i);
}
```

For-each cyklus II

- For-each neumožňuje modifikace kolekce.
- Jestli kolekci změním, nemůžeme pokračovat v iterování—dojde k vyhození

`ConcurrentModificationException`.

- Odstranění prvku a vyskočení z cyklu však funguje:

```
Set<String> set = Set.of("Donald Trump", "Barrack Obama");
for(String s: set) {
    if (s.equals("Donald Trump")) {
        set.remove(s);
        break;
    }
}
```

Iterátory

- Sekvenční procházení prvků kolekce v *neurčeném pořadí* nebo *uspořádání* (u uspořádaných kolekcí)
- Každý iterátor musí implementovat velmi jednoduché rozhraní `Iterator<E>`

Příklad s `while`

- Běžné použití pomocí `while`:

```
Set<Integer> set = Set.of(1, 2, 3);
Iterator<Integer> iterator = set.iterator();
while(iterator.hasNext()) {
    Integer element = iterator.next();
    ...
}
```

Metody iterátorů

`E next()`

- vrátí následující prvek
- `NoSuchElementException` jestli iterace nemá žádné zbývající prvky

`boolean hasNext()`

- `true` jestli iterace obsahuje nějaký prvek

`void remove()`

- odstraní prvek z kolekce
- maximálně jednou mezi jednotlivými voláními `next()`

Iterátor — příklad

Pro procházení iterátorem se dá použít i `for` cyklus:

```
Set<String> set = Set.of("Donald Trump", "Barrack Obama", "Hillary Clinton");

for (Iterator<String> iter = set.iterator(); iter.hasNext();) {
    String element = iter.next();
    if (!element.equals("Barrack Obama")) iter.remove();
}
```



Roli iterátoru plnil dříve výčet (`Enumeration`) — nepoužívat.

Replit.com demo k iterátorům (a komparátorům z příští přednášky)

- <https://replit.com/@tpitner/PB162-Java-Lecture-08-iterators-and-comparator>

Kontejnery kontejnerů

- Kromě kontejnerů obsahujících již koncové hodnoty, jsou často používané i kontejnery obsahující další kontejnery:
- Například `List<Set<Integer>>` bude obsahovat seznam množin celých čísel, tedy třeba `[[1, 4, -2], {0}, {-1, 3}]`.
- Nebo mapa, která studentům přiřazuje seznam známek `Map<Student, List<Integer>>`.

Replit.com demo ke kontejnerům kontejnerů

- <https://replit.com/@tpitner/PB162-Java-Lecture-08-map-of-lists>