Seznam, množina, iterátory

Tomáš Pitner, Radek Ošlejšek, Marek Šabo

Potomci `Collection`

Podívejme se na potomky třídy `Collection`, konkrétně:

- rozhraní `List`, implementace:
  - `ArrayList`
  - `LinkedList`
- rozhraní `Set`, implementace:
  - `HashSet`

Seznam, `List`

- něco jako dynamické pole
- každý uložený prvek má svou pozici — **číselný index**
- index je celočíselný, nezáporný, typu `int`
- možnost procházení seznamu dopředně i zpětně
- lze pracovat i s *podseznamy*:

```
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 (gif)
- rychlý přístup k prvkům dle indexu
- pomalé operace přidávání a odebírání prvků blíže k začátku seznamu (pole, v němž je seznam, se musí realokovat)

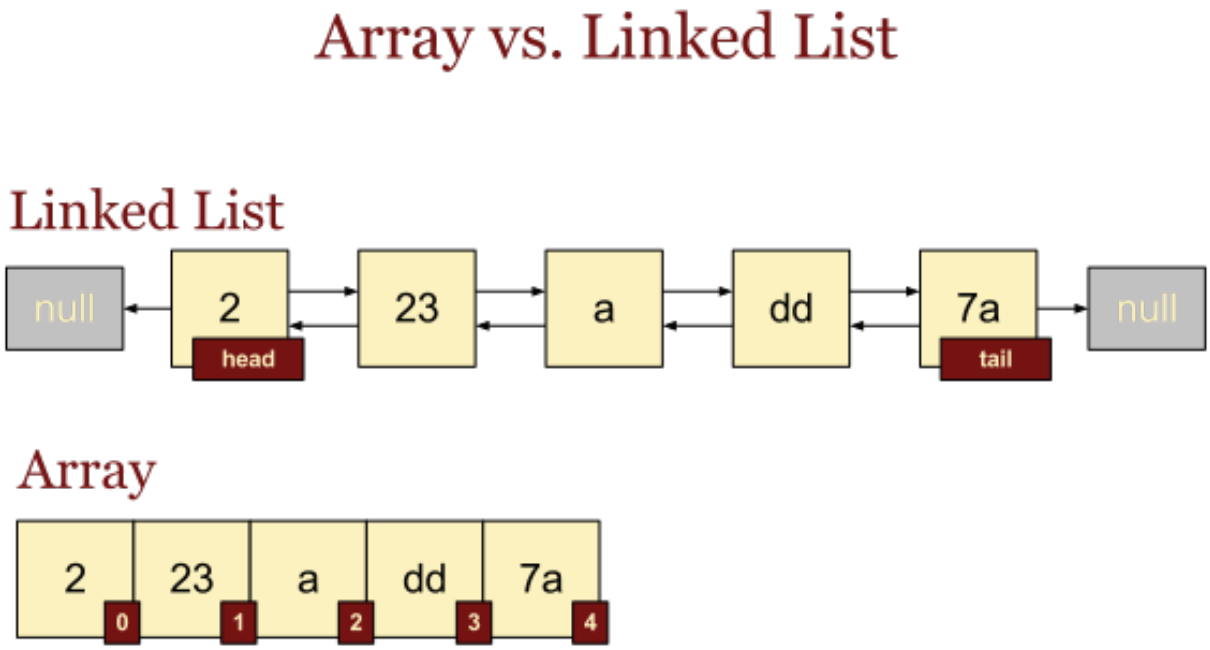
Javadoc třídy `ArrayList`

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



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

Výkonnostní porovnání seznamů

- Operace nad ArrayList vs 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**
```

Konstruktory seznamů

- `ArrayList()`
  - vytvoří prázdný seznam (s kapacitou 10 prvků)
- `ArrayList(int initialCapacity)`
  - vytvoří prázdný seznam s danou kapacitou
- `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

Statické "factory" metody:

- `List.of(elem1, elem2, ⋯)`
  - vytvoří seznam a naplní ho danými prvky
  - vrátí **nemodifikovatelnou** kolekci
- analogicky `Set.of, Map.of`
- jestli chceme kolekci modifikovatelnou, musíme vytvořit novou:

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

Na zamyšlení

- Jak udělám se seznamu typu `List` kolekci `Collection`?

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

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

```
// 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)` pro 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), 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 logicky stejné (porovnání atributů).

HashCode

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

Hash

je *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]
2    |
3    | [2,1]
```

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.
- Co když porušíme *kontrakt* metody `hashCode` (pro stejné objekty vrátí **různá** čísla)?
  - Množina přestane fungovat jako množina, bude obsahovat duplicity!

Další implmentací množiny je `LinkedHashSet` = Hash Table + Linked List

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

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 ukládájí pouze odkazy na objekty, **neukládají primitivní typy**.
- Proto používame jejich objektové protějšky — `Integer, Char, Boolean, Double`...
- Java automaticky dělá tzv. **autoboxing** — konverzi primitivního typu na objekt
- Pro zpětnou konverzi se analogicky dělá tzv. **unboxing**

```
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, užitečnější
- modifikace povolena

lambda výrazy s `forEach`

- strašidelné

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 změníme, 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>`
- 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 iterátoru 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.