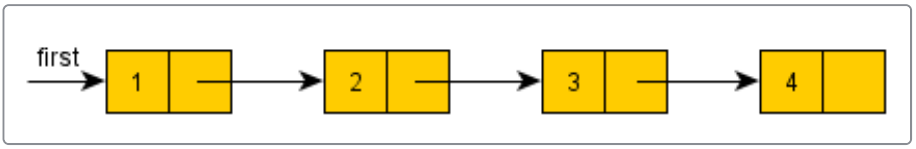
Java pro začátečníky (16) - Spojový seznam

V *desátém dílu* jsme si ukázali první kolekci – dynamické pole (*ArrayList*). Dnes si vytvoříme další kontejner – spojový seznam (*LinkedList*). Spojový seznam je jedna z nejvděčnějších datových struktur. Přestože je implementačně poměrně jednoduchý, je zároveň velmi použitelný – je základem mnoha implementací *zásobníku*, *fronty*, *grafu* a dalších datových struktur.

Dnes vytvoříme implementaci, jež nám zajistí základní funkcionalitu. Příště si ukážeme, jak seznam zobecnit pro ukládání objektů libovolného typu a jak zefektivnit jeho procházení pomocí *for-each* cyklu.

Co je to spojový seznam?

Spojový seznam je datový kontejner, který se skládá z mnoha lineárně provázaných uzlů. Každý uzel obsahuje část obsahující uloženou entitu a z ukazatel na další prvek (jednosměrně zřetězený seznam), případně i z ukazatel na prvek předchozí (obousměrně zřetězený seznam).



Jednosměrně zřetězený spojový seznam

Rozhraní (spojového) seznamu

Nejprve si specifikujme operace, které musí splňovat každý seznam. Tato abstrakce nám v případě potřeby umožní nahradit spojový seznam jinou implementací (například již zmíněným dynamickým polem).

Java

```java
/**
 * Rozhrani, jez definuje operace, ktere musi splnit kazda implementace seznamu
 * @author Pavel Micka
 */
public interface ListIface {

    /**
     * Vlozi na konec seznamu prvek
     * @param i prvek k vlozeni
     */
    public void add(int i);

    /**
     * Vrati prvek na indexu i
     * @param i index prvku
     * @return prvek na indexu i
     */
    public int get(int i);

    /**
     * Smaze prvek na indexu i
     * @param i index mazaneho prvku
     */
    public void remove(int i);

    /**
     * Dotaz na delku seznamu
     * @return delka seznamu
     */
```

```
30.     public int size();
31. }
```

Implementace

Samotný spojový seznam budeme implementovat jako jednosměrně zřetězený s ukazateli na první a poslední prvek. Ukládané hodnoty budou primitivního typu *int*.

Třídní proměnné, konstruktory a vnitřní třída Node

Java

```
01. /**
02.  * Jednosmerne zretezeny spojovy seznam
03.  * @author Pavel Micka
04.  */
05. public class MyLinkedList implements ListIface {
06. 
07.     private Node first;
08.     private Node last;
09.     private int size;
10. 
11.     /**
12.      * Konstruktor spojoveho seznamu
13.      */
14.     public MyLinkedList() {
15.         this.size = 0;
16.     }
17. 
18.     /**
19.      * Vnitrni trida reprezentujici jeden uzel spojoveho seznamu
20.      */
21.     private class Node {
22. 
23.         private int value;
24.         private Node next;
25. 
26.         private Node(int value) {
27.             this.value = value;
28.         }
29.     }
30. }
```

Třídu spojového seznamu pojmenujeme *MyLinkedList*, aby nedocházelo ke kolizi jmen s již předpřipravenou třídou *java.util.LinkedList* (*dokumentace*). Třída bude implementovat operace, které jsme si definovali v rozhraní *ListIface*.

Objekt spojového seznamu bude obsahovat ukazatel na první uzel *first*, ukazatel na poslední uzel *last* a hodnotu *size* obsahující počet prvků uložených v seznamu.

Reference na první uzel použijeme pro průchod skrz seznam (je jednosměrný). Reference na poslední prvek zjednoduší operaci ukládání nové hodnoty, protože nebudeme muset napřed proiterovat celý seznam (vkládáme nakonec). Obdobným způsobem proměnná size zjednodušší dotaz na délku seznamu.

Konstruktor obsahuje pouze jedinou operaci – inicializaci proměnné *size* na hodnotu 0.

Třída Node

Základem spojového seznamu je soukromá vnitřní třída *Node*. Tuto třídu jsme vytvořili jako vnitřní a soukromou, protože ji takto v duchu zásad zapouzdření (*encapsulation*) dokonale skryjeme před okolním světem.

Node obsahuje uloženou hodnotu *value* a referenci na další prvek *next* (*null*ový v případě, že se jedná o poslední uzel). *Gettery* a *settery* jednotlivých proměnných nejsou nevyhnutelně nutné, jelikož má k typu *Node*

přístup pouze třída *MyLinkedList* (a ta může použít přímý přístup).

Metoda add

Java

```java
/**
 * Vlozi na konec seznamu prvek
 * @param i prvek k vlozeni
 */
public void add(int i) {
    Node n = new Node(i);
    if (size == 0) {
        this.first = n;
        this.last = n;
    } else {
        this.last.next = n;
        this.last = n;
    }
    size++;
}
```

Metoda *add* přidá zadanou hodnotu na konec seznamu. Jako první musíme vytvořit nový uzel seznamu pro předávanou hodnotu. Pokud je seznam prázdný, tak se nově přidávaný uzel stane prvním (*first*) a posledním (*last*) prvkem seznamu. Pokud není seznam prázdný, tak vybereme pomocí reference *last* poslední prvek seznamu a do jeho proměnné *next* vložíme referenci na nově přidávaný uzel. Jako poslední krok v každém případě inkrementujeme proměnnou *size*.

Metoda get

Java

```java
/**
 * Vrati prvek na indexu i
 * @param i index prvku
 * @throws IndexOutOfBoundsException pokud je i vyssi nebo rovna delce seznamu
 * @throws IllegalArgumentException pokud je i zaporne
 * @return prvek na indexu i
 */
public int get(int i) {
    if (i >= size) {
        throw new IndexOutOfBoundsException("Mimo meze index:" + i + ", size:"
           + this.size);
    }
    if (i < 0) {
        throw new IllegalArgumentException("Index mensi nez 0");
    }
    Node curr = first;
    for (int j = 0; j < i; j++) {
        curr = curr.next;
    }
    return curr.value;
}
```

Metoda *get* vrátí prvek na zadaném indexu seznamu. Nejprve ošetříme chyby, které mohou nastat – předání buď příliš vysokého nebo příliš nízkého (záporného) indexu. Na obě chyby zareagujeme adekvátní nekontrolovanou výjimkou (jedná se o chybu programátora).

Nyní proiterujeme celý seznam až na daný index a vrátíme hodnotu uschovanou v daném uzlu.

Metoda remove

Java

```java
/**
 * Smaze prvek na indexu i
 * @param i index mazaneho prvku
 * @throws IndexOutOfBoundsException pokud je i vyssi nebo rovna delce seznamu
 * @throws IllegalArgumentException pokud je i zaporne
 */
public void remove(int i) {
    if (i >= size) {
        throw new IndexOutOfBoundsException("Mimo meze index:" + i + ", size:"
           + this.size);
    }
    if (i < 0) {
        throw new IllegalArgumentException("Index mensi nez 0");
    }
    if (i == 0) {
        first = first.next;
    } else {
        Node curr = first;
        for (int j = 0; j < i - 1; j++) { //najdeme predchozi
            curr = curr.next;
        }
        curr.next = curr.next.next; //a mazany prvek vynechame
        if (i == size - 1) { //pokud mazeme posledni
            last = curr;
        }
    }
    size--;
}
```

Při odstraňování prvku opět nejprve zkontrolujeme platnost zadaného indexu. Pokud mažeme první prvek, tak nám stačí změnit ukazatel na první prvek seznamu (*first*) na prvek následující. V ostatních případech nalezneme prvek, který je v seznamu před mazaným prvkem (označíme jej *curr*) a změníme jeho ukazatel na *next* na prvek, který je za mazaným prvkem (mazaný prvek vynecháme). Pokud jsme mazali poslední prvek, pak je nyní posledním prvkem *curr*. Na závěr dekrementujeme proměnnou *size*.

Pro úplnost si připomeňme, že poté, co na daný uzel zmizí poslední reference, bude odstraněn pomocí *garbage collectoru*.

Metoda size

Java

```java
/**
 * Dotaz na delku seznamu
 * @return delka seznamu
 */
public int size() {
    return this.size;
}
```

Metoda *size* pouze vrátí předpočítanou délku seznamu.

Metoda toString

Java

```java
/**
 * Klasicka toString metoda, vraci textovou reprezentaci objektu
 * @return textova reprezentace objektu
 */
@Override
public String toString() {
    StringBuilder builder = new StringBuilder();
    Node curr = first;
    for (int i = 0; i < this.size; i++) {
        builder.append(curr.value);
        builder.append(" ");
        curr = curr.next;
    }
    return builder.toString();
}
```

Nakonec ještě překryjeme metodu *toString* zděděnou z *Object*. Tato metoda vrací textovou reprezentaci daného objektu.

V této metodě využíváme *StringBuilder* (*dokumentace*), což je obdoba *ArrayListu* pro znaky. Kdybychom používali pouze operátor zřetězení +, tak bychom každým jeho použitím vytvořili nový řetězec, který bychom v zápětí zahodili (jakožto vstup nového zřetězení). Tento postup by byl velmi nehospodárný.

Kód

Takto zatím vypadá kompletní kód naší implementace spojového seznamu:

Java

```java
/**
 * Jednosmerne zretezeny spojovy seznam
 * @author Pavel Micka
 */
public class MyLinkedList implements ListIface {

    private Node first;
    private Node last;
    private int size;

    /**
     * Konstruktor spojoveho seznamu
     */
    public MyLinkedList() {
        this.size = 0;
    }

    /**
     * Vlozi na konec seznamu prvek
     * @param i prvek k vlozeni
     */
    public void add(int i) {
        Node n = new Node(i);
        if (size == 0) {
            this.first = n;
            this.last = n;
        } else {
            this.last.next = n;
            this.last = n;
        }
        size++;
    }

```

```
034.     /**
035.      * Vrati prvek na indexu i
036.      * @param i index prvku
037.      * @throws IndexOutOfBoundsException pokud je i vyssi nebo rovna delce
             seznamu
038.      * @throws IllegalArgumentException pokud je i zaporne
039.      * @return prvek na indexu i
040.      */
041.     public int get(int i) {
042.         if (i >= size) {
043.             throw new IndexOutOfBoundsException("Mimo meze index:" + i + ",
                   size:" + this.size);
044.         }
045.         if (i < 0) {
046.             throw new IllegalArgumentException("Index mensi nez 0");
047.         }
048.         Node curr = first;
049.         for (int j = 0; j < i; j++) {
050.             curr = curr.next;
051.         }
052.         return curr.value;
053.     }
054. 
055.     /**
056.      * Smaze prvek na indexu i
057.      * @param i index mazaneho prvku
058.      * @throws IndexOutOfBoundsException pokud je i vyssi nebo rovna delce
             seznamu
059.      * @throws IllegalArgumentException pokud je i zaporne
060.      */
061.     public void remove(int i) {
062.         if (i >= size) {
063.             throw new IndexOutOfBoundsException("Mimo meze index:" + i + ",
                   size:" + this.size);
064.         }
065.         if (i < 0) {
066.             throw new IllegalArgumentException("Index mensi nez 0");
067.         }
068.         if (i == 0) {
069.             first = first.next;
070.         } else {
071.             Node curr = first;
072.             for (int j = 0; j < i - 1; j++) { //najdeme predchozi
073.                 curr = curr.next;
074.             }
075.             curr.next = curr.next.next; //a mazany prvek vynechame
076.             if (i == size - 1) { //pokud mazeme posledni
077.                 last = curr;
078.             }
079.         }
080.         size--;
081.     }
082. 
083.     /**
084.      * Dotaz na delku seznamu
085.      * @return delka seznamu
086.      */
087.     public int size() {
088.         return this.size;
089.     }
090. 
091.     /**
092.      * Klasicka toString metoda, vraci textovou reprezentaci objektu
093.      * @return textova reprezentace objektu
094.      */
095.     @Override
096.     public String toString() {
```

```
097.         StringBuilder builder = new StringBuilder();
098.         Node curr = first;
099.         for (int i = 0; i < this.size; i++) {
100.             builder.append(curr.value);
101.             builder.append(" ");
102.             curr = curr.next;
103.         }
104.         return builder.toString();
105.     }
106. 
107. 
108.     /**
109.      * Vnitrni trida reprezentujici jeden uzel spojoveho seznamu
110.      */
111.     private class Node {
112. 
113.         private int value;
114.         private Node next;
115. 
116.         private Node(int value) {
117.             this.value = value;
118.         }
119.     }
120. }
```

Volání

Java | *Výstup*

```
01. /**
02.  * @param args the command line arguments
03.  */
04. public static void main(String[] args) {
05.     ListIface l = new MyLinkedList();
06. 
07.     l.add(0);
08.     l.add(10);
09.     l.add(20);
10.     l.add(30);
11.     l.add(40);
12. 
13.     //vypiseme spojovy seznam pomoci toString metody
14.     System.out.println(l);
15.     System.out.println(l.get(2)); //20
16. 
17. 
18.     l.remove(1); //odstranime prvek na prvnim indexu
19.     System.out.println(l.get(2)); //30
20. 
21.     l.remove(0);
22.     System.out.println(l);
23.     System.out.println(l.size()); //3
24. }
```

V kódu volání používáme na levé straně přiřazení rozhraní *ListIface*. Stejně tak bychom postupovali v libovolném jiném případě, kdy by vytvářená entita měla vyhovující interface.

Tomuto postupu se říká *programování proti rozhraní*. Výhody jsou poměrně jasné – pokud se rozhodneme změnit implementační třídu, tak ji vyměníme na jediném místě (tam kde vytváříme instanci) a zbytek programu může zůstat totožný. Kdybychom *programovali proti implementaci* – implementační třída by se objevovala na levých stranách přiřazení, v hlavičkách metod a podobně – tak bychom v případě výměny této třídy museli přepsat skoro celý program.

Reference vs. ukazatel

Poznámka: Bylo mi vytknuto zaměňování termínů reference a ukazatel (které nejsou v C++ totožné). Jak je nám známo, tak Java má pouze reference a nemá ukazatele. Názvoslovný problém je v tom, že javovská reference výrazně více odpovídá C++ ukazateli (oproti C++ referenci).

Pokud tedy budu mluvit v tom seriálu a ukazatelích a referencích, tak tím budu mít vždy na mysli Java referenci.

– Pavel Mička
Zakladatel a administrátor stránek Algoritmy.net