7 Liste

Nizovi su nam omogućili da rešimo ogormnu klasu problema koje do tada nismo mogli. Takodje su nam omogućili da prikupimo više literala istog tipa i smestimo ih u memoriju tako da samo sa jednim identifikatorom možemo da im pristupamo - efektivno, posmatramo ih kao na jednu promenjivu. Videli smo da na ovaj način možemo lako da pristupimo tim elementima niza i da ih menjamo, ali ogromni problem nastaje kada želimo da neke elemente izbrišemo iz niza ili pak da dodamo nove elemente u niz. Ovo pretstavlja problem jer često želimo da kreiramo programe koji obradjuju promenjive podatke: ocene u elektronskom denvniku koje mogu da se ispravljaju, cene u marketima koje mogu da se snize, nečiji broj godina itd. Pošto dosta programa zahteva ovakvo ponašanje od nizova sa kojima radi, Java programeri su kreirali **strukturu podatka** koju su nazvali **List**[1] i u Javi razlikujemo dve vrste listi:

1. ArrayList
2. LinkedList

Kako su ove liste klase, promenjive sa kojima ćemo realizovati ove liste će biti objekti, ali umesto da posmatramo na liste kao na klase, za sada možemo da ih razumemo kao novi tip podatka, koji sa sobom vuče odredjen broj ugradjenih funkcija (poput Scanner klase).

Liste pripadaju interfejsu kolekcija i nalaze se unutar `java.util` paketa. Pored toga, liste su sortirane i iterabilne kolekcije, što znači da postoji neki prirodan poredak izmedju elemenata liste.

ArrayList

Definicija 7.1 (ArrayList)**.**

ArrayList je struktura podataka koja u pozadini radi sa nizom prosledjenog tipa podatka

Šabloni kreiranja ArrayList-e su sledeći:

```
ArrayList<<omotac_tipa_podatka>> <identifikator> = new ArrayList<>();
ili
ArrayList<<omotac_tipa_podatka>> <identifikator> = new ArrayList<>(<inicijalna_duzina>);
```

Primetimo da posle `ArrayList` imamo dupliranej simbole `<<>>`. To je zato što ArrayList-a radi sa **generičkim** tipovima podataka, tj. kreirana je bez pretpostavke o tipu literala koji smeštamo u pozadinski niz. S toga, ArrayList-a zahteva da se unutar `<>` zagrada navede **omotač** tipa podatka sa kojim radimo.

Definicija 7.2 (Omotac tipa podatka)**.**

Omotac tipa podatka je klasa čije je glavno polje finalna primitivna promenjiva zadatog tipa

Suštinski, omotači nam omogućavaju da primitivne tipove podataka 'konvertujemo' u objektne, pa imamo:

- Integer -> omotač za `int`
- Double -> omotač za `double`

- Boolean -> omotač za `boolean`

Sama pozadina ovih omotača za nas nije presudna i mi ćemo ih koristit samo kada od nas to Java zahteva, poput kada kreiramo liste. Pokažimo kako možemo da kreiramo dve ArrayList-e celih brojeva:

ArrayList-Primer

```
1  package nekiPaket;
2
3  import java.util.ArrayList;
4
5  public class ArrayListe {
6
7      public static void main(String[] args) {
8          ArrayList<Integer> al1 = new ArrayList<>();
9          ArrayList<Integer> al2 = new ArrayList<>(5);
10     }
11 }
```

Ovim putem smo kreirali dve ArrayList-e koje rade sa `int` tipom podatka. Kakao liste zahtevaju neki `objekat` kao svoj generik, umesto da prosledimo primitivni tip, mi prosledjujemo `omotac` željenog primitivnog tipa. Jedina suštinska razlika je u tome što prva komanda kreira praznu listu inicijalnog kapaciteta nula i od nula elemenata, dok druga komanda kreira praznu listu inicijalnog kapaciteta 5 i od nula elemenata. Osim pozadinskog, nema razlike u funkcionalnoj mogućnosti izmedju ove dve liste; u obe liste se mogu unositi proizvoljan broj elemenata (do na dužinu najdužeg mogućeg niza).

Kada radimo sa ArrayList-ama, mi nikako ne komuniciramo sa pozadinskim nizom, već preko ugradjenih funkcija manipulišemo

njime. Pogledajmo neke od uobičajenig funkcija ArrayList-e.

1. `boolean add(E e)` -> dodaje objekat `e` klase `E` na kraj ArrayList-e i vraća uspešnost te operacije.
2. `void add(int index, E e)` -> umeće objekat `e` klase `E` na prosledjen indeks `index`.
3. `E set(int index, E e)` -> postavlja element na indeksu `index` na vrednost `e` i vraća staru vrednost koja se nalazila na tom prosledjenom indeksu.
4. `int size()` -> vraća dužinu ArrayList-e. Pandam `.length` nizovima.
5. `E get(int index)` -> vraća element na prosledjenom indeksu `index`.
6. `void clear()` -> briše sve elemente iz ArrayList-e i postavlja joj veličinu na 0.
7. `boolean isEmpty()` -> vraća odgovor na pitanje da li je ArrayList-a prazna ili nije.
8. `E remove(int index)` -> briše element iz ArrayList-e na prosledjenom indeksu `index` i vraća ga.
9. `boolean remove(Object o)` -> briše prosledjen objekat iz ArrayList-e pod uslovom da je takva operacija moguća[2] i vraća uspešnost te operacije.
10. `boolean contains(Object o)` -> vraća odgovor na pitanje da li postoji prosledjen objekat u ArrayList-i ako je takva operacija moguća[2-1].
11. `int indexOf(Object o) / int lastIndexOf(Object o)` -> vraća indeks prvog/poslednjeg pojavljivanja prosledjenog objekta u ArrayList-i ako je takva operacija moguća[2-2].
12. `String toString()` -> vraća Stringovnu reprezentaciju ArrayList-e.[3]

13. I jos mnogo drugih.

Rešimo par zadataka koji su se ticali nizova sada koristeći ArrayList-e umesto njih:

Zadaci-ArrayList

```
1  package nekiPaket;
2
3  import java.util.ArrayList;
4
5  public class ZadaciArrayList {
6
7      //Napisati funkciju koja racuna sumu elemenata
   ArrayListe
8      public static int sumaArrayListe(ArrayList<Integer>
   al) {
9          int suma = 0;
10         for (int i = 0; i < al.size(); i++) {
11             suma += al.get(i);
12         }
13         return suma;
14     }
15
16     //Alternativno sa enhanced for
17     public static int sumaArrayListe2(ArrayList<Integer>
   al) {
18         int suma = 0;
19         for (Integer elem : al) {
20             suma += elem;
21         }
22         return suma;
23     }
24
25     //Napisati funkciju koja ispisuje ArrayListu
```

```
26      public static void
    ispisiArrayListu(ArrayList<Integer> al) {
27          System.out.println(al);
28      }
29
30      //Napisati funkciju koja prikuplja cele brojeve od
    korisnika sve dok ne unese 0
31      //i smesta ih u ArrayListu
32      public static ArrayList<Integer> kreirajListu(Scanner
    sc) {
33          ArrayList<Integer> al = new ArrayList<>();
34          while (true) {
35              int unos = sc.nextInt();
36              if (unos == 0)
37                  break;
38              al.add(unos);
39          }
40          return al;
41      }
42  }
```

Primetimo da sada ne pristupamo elementima naše ArrayList-e sa uglastim zagradama `[]` već to radimo preko funkcije `get`. Nasuprot nizovima, ova funkcija vraća vrednost koja je smeštena u ArrayList-i na zadatom indeksu, što znači da nešto poput:

```
1   al.get(0) = -123;
```

nije moguće, jer ono što dobijamo sa leve strane operatora dodele nije više memorijska lokacija, već literal tipa `int`! Ukoliko žemo da promenimo vrednost prvog elementa naše liste na broj -123, to radimo sa `set` funkcijom:

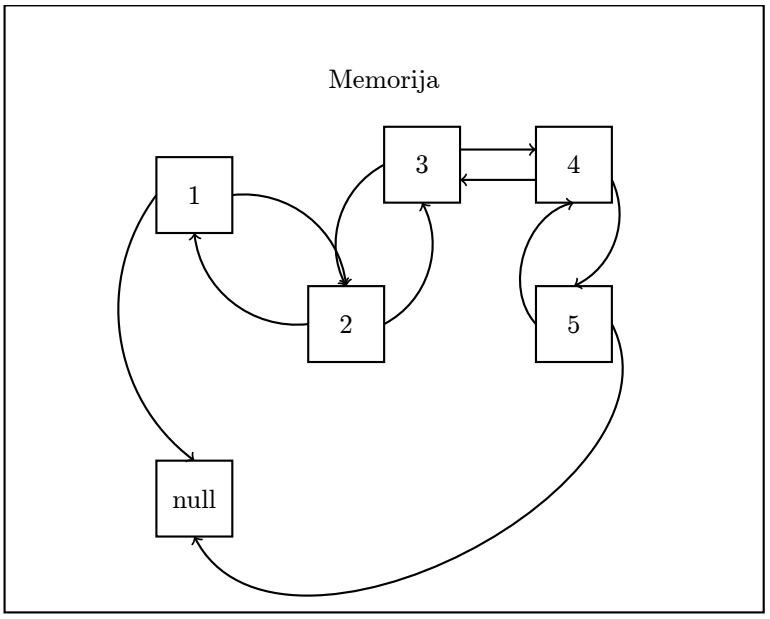
```
1   al.set(0, -123);
```

Takodje, primetimo da sada nema potrebe da pišemo naše funkcije za štampanje elemenata naše ArrayList-e kao ni korišćenje `Arrays` klase. Samim prosledjivanjem identifikatora ArrayList-e[4], Java za nas formatira i ispisuje naše elemente.

Najvažnije je da uočimo koliko jednostavno možemo da implementiramo funkciju koja od korisnika zahteva proizvoljan unos brojeva i smešta ih u ArrayList-u. Upravo ova lakoća korišćenja ArrayList-i nam pomaže da kreiramo programe koji su jako dinamički uz minimalnu dodatnu memoriju koju ArrayList-a zauzima više od bočnih nizova.
Kako ArrayList-a u pozadini radi sa običnim nizom, to zadržava svu ekspresnost u pristupanju i menjanju svojih elemenata kao što to nalaže i običan niz, dok zadržava i sve mane koje niz vuče sa sobom prilikom ubacivanja elemenata i brisanje elemenata iz sredine niza.

LinkedList

LinkedList-a je još jedna implementacija List intefejsa i pokušava da razreši probleme umetanja elemenata u sredinu liste.
LinkedList-a je implementirana kao red sa dva kraja[5]. U LinkedList-i svaki element ima svoju vrednost i pokazivač na prethodni i naredni element, s time što pokazivač na prethodni element kod prvog elementa u LinkedList-i pokazuje na `null`, kao i pokazivač na naredni element kod poslednjeg elementa u LinkedList-i. Memorijski, to možemo da prikažemo kao:



Većina funkcija koja se nalaze u ArrayList-ama nalaze se i u LinkedList-ama, poput `add`, `remove` `set` i one se ponašaju na isti način. Neke funkcije koje su ekskluzivne za LinkedList-e su:

1. `boolean offer(E e)` -> u pozadini jedino što radi je da poziva `boolean add(E e)`.[6]
2. `void push(E e)` -> postavlja na početak LinkedList-e prosledjeni element.
3. `E peekFirst() / E peekLast()` -> vraća ali ne briše prvi/poslednji element iz LinkedList-e.
4. `E poolFirst() / E poolLast()` -> vraća i briše prvi/poslednji element iz LinkedList-e
5. `E pop()` -> vraća i briše prvi element iz LinkedList-e.

Šablon kreiranja LinkedList-i je identičan šablonu kreiranja ArrayList-i s time da umesto `ArrayList`, koristimo `LinkedList` klasu.

Razlika izmedju ArrayList-e i LinkedList-e

U ovom momentu ne bi bilo loše pročitati dodatno poglavlje o složenosti algoritama u poglavlju `4 Petlje`, ali naredni odeljak je napisan tako da može da se razume i bez pročitanog dodatnog poglavlja.

Najbitnija razlika izmedju ovih dveju listi je kako rade svoje operacije. Pogledajmo njihova ponašanja za različite operacije:

1. Dodavanje elementa na kraj
   Ova operacije je vrlo jeftina za obe liste.
   - ArrayList: po potrebi produžava svoj kapacitet i u tom slučaju prekopirava ceo svoj niz u novi niz novog kapaciteta i dodaje element na kraj.
   - LinkedList: pokazivač na naredni element poslednjeg elementa liste postavlja na novi element, pokazivač na prethodni element novododatog elementa postavlja na prethodni poslednji element a pokazivač na naredni stavlja na `null`
     Zanemarujući relativno redak slučaj kada ArrayList-a prekopirava ceo svoj niz, obe ove operacije možemo smatrati instantnim.
2. Dodavanje elemenata na početak
   Ove operacija je malo skuplja za ArrayList-e jer moraju svi elementi niza da se pomeraju za jedno mesto udesno, sa potencijalnom potrebom za uvećanjem kapaciteta, dok kod LinkedList-e to može da se odradi na potpuno analogan način dodavanju elementa na kraj. S toga, ova operacija je skupla za ArrayList-u jer se izvršava linearn sa veličinom ArrayList-e dok se izvršava instantno za LinkedList-e.
3. Dodavanje elemenata u sredinu liste.
   Ova operacija je podjednako jeftina kao prethodne dve za LinkedList-e, ali za ArrayList-e podrazumeva prekopiravanje

svih elemenata koji dolaze nakon indeksa u koji umećemo element, te će se izvršiti linearno sa veličinom ArrayList-e.

4. Brisanje elemenata
   Brisanje elemenata predstavlja isti problem kao i dodavanje elemenata za ArrayList-e na isti način, dok brisanje elemenata za LinkedList-e podrazumeva samo manipulaciju dva pokazivača.
5. Pristupanje elementima
   Pristupanje elementima kod ArrayList-e se izvršava na isti način kao i pristupanje elementima niza, instantno, dok kod LinkedList-i je to nešto složeniji proces. Naime, LinkedList-a mora proći element-po-element od jednog svog početka i svaki put se pitati da li je trenutni element na zadatom indeksu ili ne. Ako nije, LinkedList-a nastavlja ovaj postupak prateći pokazivač na naredni element sve dok ne dostigne element na željenom indeksu, dakle linearno po svojoj dužini.

S toga, možemo zaključiti da ukoliko će većina operacija koje ćemo vršiti nad našim lista biti operacije:

- pristupanja elementima i dodavanjem na kraj -> onda je bolje koristiti ArrayList-u.
- dodavanjem elemenata na početak ili sredinu i brisanje elemenata -> onda je bolje koristiti LinkedList-u.

---

1. Formalno, ovo je jedan interface. Ovakva struktura podatka se javlja u skoro svim programskim jezicima, uz neku varijaciju imena (npr. u C++ jeziku, ovakva struktura se naziva vector). O interfejsima možete više pročitati u poglavlju: TODO ↩
2. Više o ovome kada budemo pričali o poredjenju objektnih tipova. ↩ ↩ ↩

3. Ovu funkciju nije neophodno eksplicitno navoditi prilikom poziva `System.out.println()`. Vise o ovome kada budemo pričali o nadjačavanju metoda. ↩
4. koji idalje predstavlja jedan pokazivač u memoriji ↩
5. Deque u nekim programskim jezicima ↩
6. Razlog zašto imamo ovako 'duplirane' funkcije je zbog nasledjivanja raznih interfejsa. Više o nasledjivanju ćemo pričati u lekciji koja se baš zove 'Nasledjivanje'. ↩