Dinamičke strukture podataka u Javi

Sadržaj

Uvod u zbirke podataka

- Zbirke (engl. *Collection*) predstavljaju podatkovnu strukturu u obliku objekta koji može sadržavati reference drugih objekata
- U Javi postoji nekoliko sučelja koja definiraju najčešće korištene zbirke podataka, a svako od sučelja ima nekoliko konkretnih aplikacija:
  - **Collection** – osnovno sučelje u hijerarhiji zbirki iz kojeg su naslijeđene zbirke Set i List
  - **Set** – sučelje zbirke koja ne može sadržavati duplikate
  - **List** – sučelje zbirke koja može sadržavati duplikate i čuva poredak elemenata
  - **Map** – sučelje zbirke koja pohranjuje parove „ključ-vrijednost”, pri čemu ključevi ne mogu imati duplikate te ne nasljeđuje sučelje **Collection**
- Ne mogu sadržavati primitivne tipove podataka (ako se pokušaju dodati u zbirku, automatski se obavlja operacija „autoboxing” koja ih pretvara u referentne tipove)

Sučelje Collection i klasa Collections

- Sučelje **Collection** sadrži skupne operacije koje se izvode nad svim objektima unutar zbirke kao što su udruživanje, brisanje i uspoređivanje objekata ili elemenata zbirke
- Zbirka iz skupine **Collection** također se može konvertirati u polje
- Sučelje **Collection** sadrži i metodu koja vraća **Iterator** objekt koji omogućava prolazak po svim elementima zbirke
- Klasa **Collections** sadrži niz statičkih metoda koje omogućavaju pretraživanje, sortiranje i druge operacije nad elementima zbirke
- Također uključuje i „wrapper” metode koje omogućavaju „sinkroniziranje” operacija nad elementima zbirke koje se koriste u višenitnom okruženju

Liste

- Poredana zbirka koja može sadržavati više referenci istog objekta
- Čuva poredak elemenata kako se u nju dodaju (prvi element će biti na početku liste, drugi odmah iza njega) te će se po istom poretku i dohvaćati iz liste
- **List** predstavlja sučelje, a najčešće korištene implementacije su **ArrayList**, **LinkedList** i **Vector**
- Klase **ArrayList** i **Vector** imaju strukturu polja čija veličina je promjenjiva
- Omogućavaju brzo lociranje elemenata u listi korištenjem indeksa, ali ne omogućavaju dobre performanse u slučaju kod umetanja elemenata usred liste
- Klasa **LinkedList** ima bolje performanse kod umetanja elemenata usred liste, ali kod pretraživanja elemenata mora slijedno prolaziti po svim elementima

Liste

- Osnovna razlika između **ArrayList** i **Vector** zbirke je u tome što su operacije unutar zbirke **Vector** „sinkronizirane” (moguće ih je koristiti u višenitnom okruženju), dok kod **ArrayList** to nije slučaj (ali ima bolje performanse)
- U praksi se najčešće koristi **ArrayList** implementacija liste
- Kod kreiranja objekta koji predstavlja klasu **ArrayList** potrebno je u izlomljenim zagradama navesti tip podatka koji će se nalaziti u listi, na primjer **„<String>**”
- Prilikom instanciranja je potrebno koristiti sučelje **List** s lijeve strane znaka jednakosti, a implementaciju s desne strane znaka jednakosti, npr.:

  List<String> listaStringova = new **ArrayList**<>();

- Time je omogućena promjena implementacije, ali zadržavanje zajedničkih metoda za sve implementacije (ne zahtijeva promjenu koda koji koristi zbirku)

Liste

- Korištenjem „Diamond operatora” (koji je uveden od Jave SE 7) nije nužno ponavljanje tipa podatka koji se nalazi unutar liste (jer se podrazumijeva, ako se ne navede):

```
List<String> listaStringova = new ArrayList<>();
```

- Kod rada s listama i dohvata njihovih elemenata moguće je koristiti „iterator” ili „foreach” petlju (koristi se puno češće nakon uvođenja u Java SE 5)
- Elementi se u listu dodaju korištenjem metode „add”, a dohvaćaju se korištenjem metode „get” i definiranjem rednog broja parametra u listi (početni indeks ima vrijednost „0”)

Primjer korištenja **ArrayList** implementacije (1)

```
String[] poljeStringova = {"PROLJEĆE", "LJETO", "JESEN", "ZIMA", "ZIMA"};
List<String> listaStringova = new ArrayList<>();

//foreach petlja
for(String doba : poljeStringova) {
   listaStringova.add(doba);
}

Iterator<String> iterator = listaStringova.iterator();

while(iterator.hasNext()) {
   System.out.println(iterator.next());
}
```

Ispis:

PROLJEĆE
LJETO
JESEN
ZIMA
ZIMA

Primjer korištenja **ArrayList** implementacije (2)

```
String[] poljeStringova = {"PROLJEĆE", "LJETO", "JESEN", "ZIMA", "ZIMA"};
List<String> listaStringova = new ArrayList<>();

for(String doba : poljeStringova) {
   listaStringova.add(doba);
}

for(String doba : listaStringova) {
   System.out.println(doba);
}
```

Ispis:

PROLJEĆE
LJETO
JESEN
ZIMA
ZIMA

Primjer korištenja **LinkedList** implementacije

```
String[] poljeStringova = {"PROLJEĆE", "LJETO", "JESEN", "ZIMA", "ZIMA"};
List<String> listaStringova = new LinkedList<>();
for(String doba : poljeStringova) {
   listaStringova.add(doba);
}

for(String doba : listaStringova) {
   System.out.println(doba);
}

System.out.println();

String pop = ((LinkedList<String>)listaStringova).pop();
System.out.println(pop);
String prvi = ((LinkedList<String>)listaStringova).getFirst();
System.out.println(prvi);
```

Ispis:

```
PROLJEĆE
LJETO
JESEN
ZIMA
ZIMA

PROLJEĆE
LJETO
```

Korištenje metoda iz klase **Collections**

- Sadrži niz statičkih metoda koje uključuju algoritme visokih performansi za rad s elementima zbirke
- Primjeri metoda:
  - **sort** – sortira elemente liste
  - **binarySearch** – koristi algoritam binarnog pretraživanja zbirke kod traženja određenog elementa
  - **reverse** – obrnuto sortira elemente u listi
  - **shuffle** – miješa elemente prema slučajnom redoslijedu
  - **fill** – popunjava zbirku
  - **copy** – kopira element jedne zbirke u drugu
  - **min** – vraća najmanji element iz zbirke
  - **max** – vraća najveći element iz zbirke
  - **addAll** – dodaje elemente iz jedne zbirke u drugu
  - **frequency** – određuje koliko puta se neki element pojavljuje unutar zbirke
  - **disjoint** – provjerava sadrže li dvije zbirke iste objekte

Korištenje metoda iz klase **Collections**

- Primjer korištenja:

```
String[] poljeStringova = {"PROLJEĆE", "LJETO", "JESEN", "ZIMA", "ZIMA"};

List<String> listaStringova = Arrays.asList(poljeStringova);
System.out.println(listaStringova);

Collections.sort(listaStringova);
System.out.println(listaStringova);

int broj = Collections.frequency(listaStringova, "LJETO");
System.out.println(broj);
```

Ispis:

```
[PROLJEĆE, LJETO, JESEN, ZIMA, ZIMA]
[JESEN, LJETO, PROLJEĆE, ZIMA, ZIMA]
1
```

Setovi

- Set je zbirka koja ne čuva poredak elemenata (kakav je bio kod umetanja elemenata) i sadržava samo po jednu referencu određenog objekta (bez duplikata)
- Umetanje elementa koji već postoji zamjenjuje stari element iz seta
- Elementi se u **Set** dodaju metodom „add”, a dohvaćaju slučajnim redoslijedom pomoću **Iteratora** ili „foreach” petlje
- Najčešće implementacije sučelja **Set** su **HashSet** i **TreeSet**
- Osim sučelja **Set** postoji i sučelje **SortedSet** koje omogućava sortiranje elemenata i sadrži metode koje dohvaćaju sortirane elemente, npr. „first” i „last”
- Implementacija **TreeSet** osim sučelja **Set** implementira i sučelje **SortedSet**
- Objekt klase koja implementira sučelje Set može se kreirati na sljedeći način:

```
Set<String> setStringova = new HashSet<>();
```

Primjer korištenja HashSet implementacije

```
String[] poljeStringova = {"PROLJEĆE", "LJETO", "JESEN", "ZIMA", "ZIMA"};
Set<String> setStringova = new HashSet<>();

for(String doba : poljeStringova) {
   setStringova.add(doba);
}

for(String doba : setStringova) {
   System.out.println(doba);
}
```

Ispis:
JESEN
LJETO
PROLJEĆE
ZIMA

Mape

- Zbirke koje povezuju ključeve s vrijednostima
- Ključevi u mapi moraju biti jedinstveni (što se postiže korištenjem seta), a vrijednosti ne moraju biti jedinstvene (različiti ključevi mogu imati iste vrijednosti)
- Najčešće implementacije sučelja **Map** su **Hashtable**, **HashMap** i **TreeMap**
- Klase **Hashtable** i **HashMap** za spremanje podataka koriste *hash* tablicu (koja sadrži *sažetke* objekata), a **TreeMap** koristi strukturu stabla
- Kako **TreeMap** implementira sučelje **SortedMap**, elemente je u mapi moguće i sortirati
- Kod kreiranja objekta klase koja predstavlja implementaciju mape, potrebno je navesti tip koji predstavlja ključ i tip koji predstavlja vrijednost:

Map<**Integer, Integer**> mapaUnesenihBrojeva = **new HashMap<>();**

Primjer korištenja HashMap implementacije

```
Map<Integer, Integer> mapaUnesenihBrojeva = new HashMap<>();
Scanner unos = new Scanner(System.in);
for(int i = 0; i < 10; i++) {
   System.out.print("Unesite broj: ");
   Integer broj = unos.nextInt();
   if(mapaUnesenihBrojeva.containsKey(broj)) {
      Integer kolicina = mapaUnesenihBrojeva.get(broj);
      kolicina += 1;
      mapaUnesenihBrojeva.put(broj, kolicina);
   }
   else {
      mapaUnesenihBrojeva.put(broj, 1);
   }
}
System.out.println(mapaUnesenihBrojeva);
unos.close();
```

```
Unos:
Unesite broj: 8
Unesite broj: 9
Unesite broj: 6
Unesite broj: 5
Unesite broj: 5
Unesite broj: 6
Unesite broj: 6
Unesite broj: 3
Unesite broj: 2
Unesite broj: 1
Ispis:
{1=1, 2=1, 3=1, 5=2,
6=3, 8=1, 9=1}
```

Primjer korištenja HashMap implementacije

- Metodom „containsKey” moguće je provjeriti sadrži li mapa traženi ključ pa tek nakon toga dodavati nove elemente, ako ključ ne postoji
- Dodavanjem ključa koji već postoji u mapi **izbacuje se stara vrijednost i dodaje nova**, programski kod **ne dojavljuje pogrešku**
- Pomoću metode „get” se vraća vrijednost iz mape za zadani ključ, a pomoću metode „put” se dodaje novi par „ključ-vrijednost” u mapu
- Postoji i metoda „keySet” koja iz mape dohvaća set ključeva koji je moguće iskoristiti za dohvat svih elemenata iz mape:

```
for(Integer key : mapaUnesenihBrojeva.keySet()) {
    System.out.println(key + " " + mapaUnesenihBrojeva.get(key));
}
```

Enumeracije

- Enumeracija je tip podatka koja sadrži pobrojani niz konstanti
- To su referentni tipovi koji mogu imati konstruktor i varijable
- Mogu se koristiti umjesto skupa konstanti unutar „switch-case” konstrukta
- Pomoću metode „values” moguće je dohvatiti sve vrijednosti iz enumeracije
- Za enumeracije vrijede sljedeća pravila:
  - Konstante u enumeracijama su implicitno označene modifikatorima **final** i **static**
  - Nije moguće kreirati objekt koji predstavlja enumeraciju, već je moguće koristiti samo predefinirane vrijednosti
- Primjer jednostavne enumeracije:

```
public enum GodisnjeDoba {
   PROLJECE, LJETO, JESEN, ZIMA
}
```

Enumeracije

- Primjer složenije enumeracije:

```
public enum StatusObrade {

    USPJESNA_OBRADA(1, "Uspješno obrađene sve transakcije"),
    TIMEOUT(2, "Neuspješna obrada, isteklo maksimalno vrijeme trajanja"),
    NEISPRAVNI_PODACI(3, "Neuspješna obrada, neispravni podaci u transakcijama");

    private Integer kod;
    private String opis;

    private StatusObrade(Integer kod, String opis) {
        this.kod = kod;
        this.opis = opis;
    }
    //getter metode za kod i opis
}
```

Primjer korištenja enumeracije

```
for(GodisnjeDoba godisnjeDoba : GodisnjeDoba.values()) {
   System.out.println(godisnjeDoba);
}

StatusObrade status = StatusObrade.USPJESNA_OBRADA;
switch(status) {
        case USPJESNA_OBRADA:
                System.out.println("Poruka1");
                break;
        case TIMEOUT:
                System.out.println("Poruka2");
                break;
        case NEISPRAVNI_PODACI:
                System.out.println("Poruka3");
                break;
}
```

Tokovi u Java SE 8

- Java SE 8 uvodi koncept tokova (engl. *Streams*) koji su slični iteratorima za manipuliranje elementima zbirki
- Omogućavaju paralelno obrađivanje podataka čime se znatno poboljšavaju performanse
- Svaka zbirka ili polje ima mogućnost pozivanja metode „stream” i nad dobivenim tokom obavljati operacije
- Tokovi služe za obradu elemenata zbirke kroz nekoliko procesnih koraka (metoda) koji se nazivaju i cjevovod toka (engl. *stream pipeline*)
- Sastoje se od međuoperacija (engl. *Intermediate operation*) i završnih operacija (engl. *Terminal operation*)
- Međuoperacije se pišu prije, ali se izvršavaju tek kad se obavi i završna operacija

Međuoperacije i završne operacije u tokovima

- Primjeri međuoperacija:
  - **filter** – rezultira tokom koji sadrži one elemente prema zadanim kriterijima
  - **distinct** – rezultira tokom koji sadrži samo jedinstvene elemente (bez duplikata)
  - **limit** – rezultira tokom koji sadrži samo određeni broj elemenata iz početnog toka
  - **map** – rezultira tokom u kojem je svaki od elemenata „mapiran” na drugu vrijednost, na primjer, cjelobrojni elementi mogu biti „mapirani” na svoje kvadrate vrijednosti
  - **sorted** – rezultira tokom u kojem su sortirani elementi
- Primjeri završnih operacija:
  - **forEach** – obavlja procesiranje svakog elementa u toku
  - **average** – izračunava srednju vrijednost elemenata u toku koji sadrži numeričke vrijednosti
  - **count** – vraća broj elemenata u toku
  - **max** – vraća maksimalnu vrijednost u toku s numeričkim vrijednostima

Međuoperacije i završne operacije u tokovima

- **min** – vraća minimalnu vrijednost iz toka koji sadrži numeričke vrijednosti
- **reduce** – određuje jedinstvenu vrijednost toka korištenjem lambda funkcije (npr. suma elemenata)
- **collect** – kreira novu zbirku elemenata koja sadrži samo rezultate obrada elemenata u toku
- **toArray** – kreira polje elemenata koje sadrži samo rezultate obrada elemenata u toku
- **findFirst** – traži prvi element predstavljen objektom Optional u toku nakon izvršavanja međuoperacija iz kojeg je pomoću „get” metode moguće dohvatiti vrijednost elementa
- **findAny** – traži bilo koji element predstavljen objektom Optional u toku nakon izvršavanja međuoperacija iz kojeg je pomoću „get” metode moguće dohvatiti vrijednost elementa
- **anyMatch** – određuje ispunjava li bilo koji element uvjete definirane u međuoperacijama
- **allMatch** – određuje ispunjavaju li svi elementi u toku uvjete definirane u međuoperacijama

Primjer korištenja IntStream operacija

- Specijalizirani tok **IntStream** za rad sa cjelobrojnim vrijednostima može se koristiti na sljedeći način:

```
int[] brojevi = {8, 10, 4, 9, 5, 7, 1, 3};

System.out.print("Početne vrijednosti: ");
IntStream.of(brojevi)
.forEach(vrijednost -> System.out.print(vrijednost + " "));

System.out.println();
System.out.print("Broj elemenata: ");
System.out.print(IntStream.of(brojevi).count());
```

Primjer korištenja IntStream operacija

```
System.out.println();
System.out.print("Najveći element: ");
System.out.print(IntStream.of(brojevi).max().getAsInt());

System.out.println();
System.out.print("Suma elemenata: ");
System.out.print(IntStream.of(brojevi).reduce(0, (x, y) -> x + y));

System.out.println();
System.out.print("Sortirani parni brojevi: ");
IntStream.of(brojevi).filter(vrijednost -> vrijednost % 2 == 0)
   .sorted()
   .forEach(vrijednost -> System.out.print(vrijednost + " "));
```

Ispis:

```
Početne vrijednosti: 8 10 4 9
5 7 1 3
Broj elemenata: 8
Najveći element: 10
Suma elemenata: 47
Sortirani parni brojevi: 4 8 10
```

Primjer korištenja operacija tokova nad listom

- Metodu „stream” moguće je koristiti i nad objektima koji predstavljaju polja ili zbirke, npr. nad listom:

```
TekuciRacun prviRacun = new TekuciRacun("12345", new BigDecimal(1000.00));
TekuciRacun drugiRacun = new TekuciRacun("45678", new BigDecimal(2000.00));
TekuciRacun treciRacun = new TekuciRacun("56789", new BigDecimal(3000.00));
List<TekuciRacun> listaRacuna = new ArrayList<>();
listaRacuna.add(prviRacun);
listaRacuna.add(drugiRacun);
listaRacuna.add(treciRacun);

listaRacuna.stream().filter(vrijednost -> (vrijednost.getStanje().doubleValue() >= 2000))
.forEach(System.out::println);
```

Ispis:

45678(2000)
56789(3000)

Primjer korištenja operacija tokova nad listom

- Korištenjem operacije „filter” moguće je definirati lambda izraz koji omogućava filtriranje elemenata liste po zadanim kriterijima
- Nakon operacije filtriranja slijedi operacija „forEach” kojom je omogućeno pozivanje metode „System.out.println” pomoću operatora „::”
- Osim pisanja izraza unutar „forEach” metode, moguće je i prije kreirati objekt funkcionalnog sučelja „Predicate”, te nakon toga predati u metodu „filter”:

```
Predicate<TekuciRacun> stanjeIznadDvijeTisuceKuna =
        e -> (e.getStanje().doubleValue() >= 2000);

listaRacuna.stream().filter(stanjeIznadDvijeTisuceKuna)
        .forEach(System.out::println);
```

Nove funkcionalnosti sa zbirkama u Javi 9

- U Javi 9 su uvedene nove mogućnosti kreiranja „immutable” listi (koje se ne mogu mijenjati) korištenjem „of” metode:

```
List<String> immutableList = List.of("Prvi","Drugi","Treći");
```

- Nakon kreiranja liste naknadne promjene njenog sadržaja rezultiraju bacanjem iznimke „UnsupportedOperationException”:

```
immutableList.add("Četvrti");
```

- Na sličan način je moguće kreirati i mapu:

```
Map<Integer, String> immutableMap = Map.of(1, "Prvi", 2, "Drugi", 3, "Treći");
```

Nove funkcionalnosti sa *streamovima* u Javi 9

- Java 9 je uvela i nove načine rada sa *streamovima*, na primjer, ispis svih vrijednosti unutar „IntStreama" moguće je obaviti na sljedeći način:

```
IntStream.iterate(1, i -> i < 100, i -> i + 1).forEach(System.out::println);
```

- Osim toga je moguće nad *streamovima* koristiti metodu „takeWhile" za ispis samo onih vrijednosti koje ispunjavaju zadani uvjet:

```
Stream.of(1,2,3,4,5,6,7,8,9,10).takeWhile(i -> i < 5 ).forEach(System.out::println);
```

- Na sličan način funkcionira metoda „dropWhile" koja kao rezultat ostavlja samo one vrijednosti koji ne ispunjavaju zadani uvjet:

```
Stream.of(1,2,3,4,5,6,7,8,9,10).dropWhile(i -> i < 5 ).forEach(System.out::println);
```

Pitanja?