DIZAJN I RAZVOJ SOFTVERA

Rad sa kolekcijama - Nastavak

Klasa Collections

- Klasa **`Collections`** obezbjeđuje nekoliko efikasnih algoritama za manipulaciju elementima kolekcije. Ovi algoritmi su implementirani kao statičke metode. Najčešće korišćene metode su date u tabeli ispod.

Metoda	Opis
`sort`	Sortiranje elemenata liste.
`binarySearch`	Traženje elementa liste.
`reverse`	Obrtanje redosljeda elemenata liste.
`shuffle`	Slučajno raspoređivanje elemenata (miješanje) liste.
`fill`	Upisivanje određene reference u svaki element liste.
`copy`	Kopiranje reference iz jedne liste u drugu.
`min`	Vraća minimalan element kolekcije.
`max`	Vraća maksimalan element kolekcije.
`addAll`	Nadovezivanje elemenata niza na kolekciju.

Primjer sa klasom Collections

```
import java.util.Collections;
import java.util.ArrayList;
import java.util.List;
import java.util.Arrays;

public class CollectionsTest {

   public static void main(String[] args) {
      Integer niz[] = {3, 4, 11, 9, 7, 4, 17, 1};
      List< Integer > lista = new ArrayList< Integer >(Arrays.asList(niz));
      System.out.printf("Početna lista:\n%s ", lista);

      System.out.printf("\nMinimum liste %d, maksimum %d.",
           Collections.min(lista), Collections.max(lista));

      Collections.sort(lista);
      System.out.printf("\nLista nakon rast. sortiranja:\n%s", lista);
      Collections.sort(lista, Collections.reverseOrder());
      System.out.printf("\nLista nakon opad. sortiranja:\n%s", lista);

      Collections.shuffle(lista);
      System.out.printf("\nLista nakon miješanja:\n%s", lista);

      Collections.addAll(lista, 5, 13);
      System.out.printf("\nLista nakon dodavanja:\n%s", lista);

      Collections.fill(lista, 7);
      System.out.printf("\nLista nakon zamene elemenata\n%s", lista);
   }
}
```

Ispis

```
Početna lista:
[3, 4, 11, 9, 7, 4, 17, 1]
Minimum liste 1, maksimum 17.

Lista nakon rast. sortiranja:
[1, 3, 4, 4, 7, 9, 11, 17]
Lista nakon opad. sortiranja:
[17, 11, 9, 7, 4, 4, 3, 1]

Lista nakon miješanja:
[9, 11, 3, 1, 7, 4, 17, 4]

Lista nakon dodavanja:
[9, 11, 3, 1, 7, 4, 17, 4, 5, 13]

Lista nakon zamjene elemenata
[7, 7, 7, 7, 7, 7, 7, 7, 7, 7]
```

Komentari primjera

- Metoda `sort` sortira elemente liste, pri čemu klasa elemenata liste mora implementirati **`Comparable`** interfejs. Redosljed sortiranja je definisan metodom **`compareTo`**, koja je deklarisana u ovom interfejsu.
- Metoda `sort` može imati i drugi parametar koji će odrediti alternativno uređenje elemenata. U našem primjeru, sortiranje u opadajući redosljed se dobija naredbom

```
Collections.sort(lista, Collections.reverseOrder());
```

- Statička metoda **`reverseOrder()`** klase `Collections` vraća **`Comparator`** objekat koji uređuje elemente u suprotan redosljed.
- Za objekte koji ne implementiraju interfejs `Comparable`, moramo sami kreirati `Comparator` objekte!
- U nastavku ćemo objasniti dva načina poređenja korisnički definisanih objekata. Prvi je baziran na implementaciji interfejsa `Comparable`, drugi na kreiranju `Comparator` objekta.

Klasa Osoba i interfejs Comparable

```
public class Osoba implements Comparable< Osoba > {
   String imePrezime;
   double visina, tezina;

   public Osoba(String imePrez, double vis, double tez) {
      setImePrezime(imePrez);
      setVisina(vis);
      setTezina(tez);}
   public String getImePrezime() {return imePrezime;}
   public void setImePrezime(String imePrez) {imePrezime = imePrez;}
   public double getVisina() {return visina;}
   public void setVisina(double vis) {visina = vis;}
   public double getTezina() {return tezina;}
   public void setTezina(double tez) {tezina = tez;}

   public String toString() {
      return String.format("%s, visina %.1f cm, težina %.1f kg",
                           getImePrezime(), getVisina(), getTezina());}

   @Override
   public int compareTo(Osoba drugi) {
      double vis1 = this.getVisina();
      double tez1 = this.getTezina();
      double vis2 = drugi.getVisina();
      double tez2 = drugi.getTezina();
      if((vis1 == vis2) && (tez1 == tez2))   return 0;
      else if((vis1 > vis2) || ((vis1 == vis2) && (tez1 > tez2))) return 1;
      else return -1;
   }
}
```

`Comparable` je generički interfejs (obratite pažnju na zagrade < >) iz paketa `java.lang` sa jednom metodom **`compareTo`** koja služi za poređenje objekta koji poziva metodu sa objektom parametrom po kriterijumu koji mi definišemo. Metoda vraća negativan cio broj ako je objekat parametar veći, pozitivan cio broj ako je objekat parametar manji i 0 ako su objekat parametar i objekat koji poziva metodu isti.

Poredak elemenata koji definiše metoda `compareTo` se naziva prirodni poredak (eng. *natural ordering*), dok se ta metoda naziva metoda prirodnog poređenja (eng. *natural comparison method*).

Klasa Osoba i interfejs Comparable

```
import java.util.*;

public class OsobaSort {

   public static void main(String[] args) {
      List< Osoba > lista = new ArrayList< Osoba >();

      lista.add(new Osoba("Dalibor Katić", 178.4, 89.4));
      lista.add(new Osoba("Ana Jović", 171.2, 65.1));
      lista.add(new Osoba("Vesna Milić", 168.9, 59.7));
      lista.add(new Osoba("Nikolina Bulović", 184.3, 86.0));
      lista.add(new Osoba("Jovica Zvrko", 178.4, 79.1));

      System.out.println("Nesortirana lista:");
      for(Osoba x: lista)
         System.out.println(x);

      Collections.sort(lista);

      System.out.println("\nSortirana lista:");
      for(Osoba x: lista)
         System.out.println(x);
   }
}
```

Ispis

```
Nesortirana lista:
Dalibor Katić, visina 178.4 cm, težina 89.4 kg
Ana Jović, visina 171.2 cm, težina 65.1 kg
Vesna Milić, visina 168.9 cm, težina 59.7 kg
Nikolina Bulović, visina 184.3 cm, težina 86.0 kg
Jovica Zvrko, visina 178.4 cm, težina 79.1 kg

Sortirana lista:
Vesna Milić, visina 168.9 cm, težina 59.7 kg
Ana Jović, visina 171.2 cm, težina 65.1 kg
Jovica Zvrko, visina 178.4 cm, težina 79.1 kg
Dalibor Katić, visina 178.4 cm, težina 89.4 kg
Nikolina Bulović, visina 184.3 cm, težina 86.0 kg
```

Interfejs Comparator

- **`Comparator`** je generički interfejs iz paketa `java.util` koji se koristi za sortiranje objekata korisničkih klasa. Pruža dvije bitne prednosti u odnosu na `Comparable` interfejs:
  - omogućava poređenje i sortiranje po više kriterijuma,
  - ne zahtijeva izmjenu originalne klase, odnosno omogućava poređenje za klase čiju realizaciju ne možemo modifikovati.
- Posmatrajmo klasu `Osoba` iz prethodnog primjera. Metoda `compareTo` omogućava da sortiramo listu tipa `Osoba` po unaprijed definisanom kriterijumu i to je jedino sortiranje koje možemo postići na ovaj način. Čak ni sortiranje u opadajući redosljed nije omogućeno.
- `Comparator` interfejs omogućava proizvoljan broj načina sortiranja elemenata predmetnog tipa. Za svaki način sortiranja, potrebno je da definišemo klasu koja implementira generički interfejs `Comparator< Tip >`, u okviru koje ćemo realizovati metodu **`compare`**.
- Ova metoda ima dva parametra - objekte predmetnog tipa koje poredi, i vraća negativan cio broj ako je prvi parametar objekat manji od drugog, pozitivan cio broj ako je prvi parametar veći od drugog, i 0 ako su jednaki.

Klasa Osoba i interfejs Comparator

```
public class Osoba {

   String imePrezime;
   double visina, tezina;

   public Osoba(String imePrez, double vis, double tez){
      setImePrezime(imePrez);
      setVisina(vis);
      setTezina(tez); }

   public String getImePrezime() {
      return imePrezime; }

   public void setImePrezime(String imePrez) {
      imePrezime = imePrez; }

   public double getVisina() {
      return visina; }

   public void setVisina(double vis) {
      visina = vis; }

   public double getTezina() {
      return tezina; }

   public void setTezina(double tez) {
      tezina = tez; }

   public String toString(){
      return String.format("%s, visina %.1f cm, težina %.1f kg",
           getImePrezime(), getVisina(), getTezina());
   }
}
```

Klasa Osoba i interfejs Comparator

```
import java.util.Comparator;
public class OsobaRastuci implements Comparator< Osoba > {
   public int compare(Osoba prvi, Osoba drugi) {
      double vis1 = prvi.getVisina();
      double tez1 = prvi.getTezina();
      double vis2 = drugi.getVisina();
      double tez2 = drugi.getTezina();
      if((vis1 == vis2) && (tez1 == tez2))  return 0;
      else if((vis1 > vis2) || ((vis1 == vis2) && (tez1 > tez2)))  return 1;
      else  return -1;
   }
}
```

```
import java.util.Comparator;
public class OsobaOpadajuci implements Comparator< Osoba > {
   public int compare(Osoba prvi, Osoba drugi) {
      double vis1 = prvi.getVisina();
      double tez1 = prvi.getTezina();
      double vis2 = drugi.getVisina();
      double tez2 = drugi.getTezina();
      if((vis1 == vis2) && (tez1 == tez2))  return 0;
      else if((vis1 > vis2) || ((vis1 == vis2) && (tez1 > tez2)))  return -1;
      else  return 1;
   }
}
```

Klasa Osoba i interfejs Comparator

```
import java.util.*;

public class OsobaSort {

   public static void main(String[] args) {
      LinkedList< Osoba > lista = new LinkedList< Osoba >();

      lista.add(new Osoba("Dalibor Katić", 178.4, 89.4));
      lista.add(new Osoba("Ana Jović", 171.2, 65.1));
      lista.add(new Osoba("Vesna Milić", 168.9, 59.7));
      lista.add(new Osoba("Nikolina Bulović", 184.3, 86.0));
      lista.add(new Osoba("Jovica Zvrko", 178.4, 79.1));

      Collections.sort(lista, new OsobaRastuci());

      System.out.println("Sortirana lista (rastući):");
      for(Osoba x: lista)
         System.out.println(x);

      Collections.sort(lista, new OsobaOpadajuci());

      System.out.println("\nSortirana lista (opadajući):");
      for(Osoba x: lista)
         System.out.println(x);
   }
}
```

Ispis

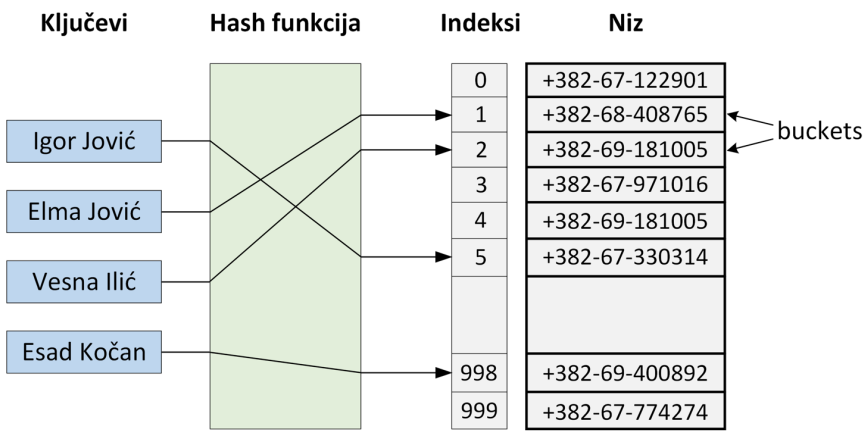
```
Sortirana lista (rastući):
Vesna Milić, visina 168.9 cm, težina 59.7 kg
Ana Jović, visina 171.2 cm, težina 65.1 kg
Jovica Zvrko, visina 178.4 cm, težina 79.1 kg
Dalibor Katić, visina 178.4 cm, težina 89.4 kg
Nikolina Bulović, visina 184.3 cm, težina 86.0 kg

Sortirana lista (opadajući):
Nikolina Bulović, visina 184.3 cm, težina 86.0 kg
Dalibor Katić, visina 178.4 cm, težina 89.4 kg
Jovica Zvrko, visina 178.4 cm, težina 79.1 kg
Ana Jović, visina 171.2 cm, težina 65.1 kg
Vesna Milić, visina 168.9 cm, težina 59.7 kg
```

Skupovi

- **Skup** (eng. *set*) predstavlja kolekciju jedinstvenih elemenata (nema ponavljanja elemenata!).
- Postoji nekoliko `Set` implementacija, od kojih su najpopularniji `HashSet` i `TreeSet`.
- `HashSet` smješta elemente u hash tabeli ili hash mapi, dok `TreeSet` smješta elemente u stablu.
- U `TreeSet`-u, elementi su sortirani!
- Pošto `HashSet` i `TreeSet` ne mogu sadržati duplikate elemenata, ponavljanja se onemogućavaju tokom kreiranja.
- Zbog značaja hash tabele u računarstvu, ovdje ćemo joj posvetiti malo prostora.

Hash tabela

- U računarstvu, hash tabela je struktura podataka koja implementira **asocijativni niz** (eng. *associative array*), apstraktni tip podatka koji vrši efikasno preslikavanje ključeva (npr. imena i prezimena Osoba) u vrijednosti (npr. telefonske brojeve).
- Za ovo preslikavanje se koristi hash funkcija koja transformiše ključ u indeks (ili hash kôd) niza gdje je smještena tražena vrijednost. Pojedinačne pozicije u nizu vrijednosti se zovu **buckets** ili slots.
- Tokom pretrage, potrebno je izračunati indeks i direktno pristupiti tom elementu u nizu, bez obzira na veličinu niza.
- Vrijeme pretrage ne zavisi od broja veličine tabele!
- Zbog ove karakteristike, kao i činjenice da su efikasnije od ostalih “brzih” tipova podataka (npr. stabala za pretragu), hash tabele su vrlo popularan izbor u svim vrstama softvera.



Implementacija telefonskog imenika uz pomoć hash tabele

Hash tabela - Ograničenja

- Idealno, hash funkcija jednoznačno preslikava ključeve u indekse (jedan ključ – jedan indeks). Međutim, ovo u praksi najčešće nije slučaj.
- Većina implementacija hash tabela podrazumijeva da su **hash kolizije** (parovi različitih ključeva sa istim indeksima) normalna pojava, i na neki način prevazilaze ovaj problem.
- Hash tabela koja implementira telefonski imenik (slajd 12) podrazumijeva da nema kolizije. U slučaju kolizije, pored brojeva telefona bismo morali čuvati i imena osoba u tabeli, radi jednoznačnog određivanja para osoba-telefon.
- Popularno razrješavanje kolizije pruža **metoda ulančavanja** (eng. *chaining*). Suština metode je da se formira povezana lista svih vrijednosti u tabeli na koje ukazuju sinonimi (ključevi koji se preslikavaju u isti indeks).
- Vrijeme potrebno za pretragu u tom slučaju jednako je vremenu određivanja indeksa pomoću hash funkcije (konstantno) plus vrijeme potrebno da se prođe kroz povezanu listu.

hashCode metoda klase Object

- `hashCode` metoda vraća hash kôd objekta.
- Uslovi koje generalno ova metoda treba da ispuni su:
  - Koliko god puta da se pozove nad istim objektom tokom izvršenja aplikacije, `hashCode` mora da vrati isti cio broj. U različitim izvršenjima iste aplikacije, ovaj broj ne mora biti isti.
  - Ako su dva objekta ista prema `equals` metodi, `hashCode` metoda mora da vrati isti kôd za oba objekta.
  - Dozvoljeno je da `hashCode` vrati isti kôd za dva objekta koja su različita prema `equals` metodi. Ipak, ovo nije poželjna karakteristika, jer pogoršava performanse pretrage hash tabele.
- `hashCode` metoda klase `Object` vraća različite cjelobrojne kôdove za različite objekte. Ova metoda može bazirati svoju implementaciju na adresi objekta u memoriji, koja je jedinstvena za svaki objekat.
- Poznata razvojna okruženja nude podrazumijevane realizacije `hashCode` metode za korisničke klase. Na sledećem slajdu dajemo jedan primjer za okruženje Eclipse.

hashCode implementacija u Eclipse-u

- Imamo klasu `Test` koja za podatke ima **`double x`** i **`String s`**.

- Realizacija `hashCode` metode koju nam nudi Eclipse je data ispod:

```
public int hashCode() {
   final int prime = 31;
   int result = 1;
   result = prime * result + ((s == null) ? 0 : s.hashCode());
   long temp = Double.doubleToLongBits(x);
   result = prime * result + (int) (temp ^ (temp >>> 32));
   return result;
}
```

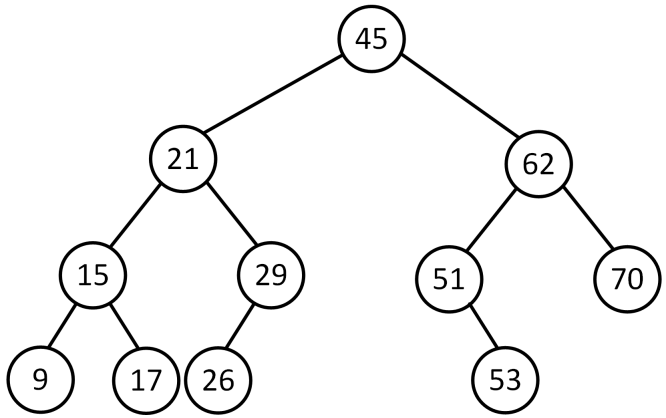
Klasa HashSet

- **`HashSet`** je klasa iz paketa `java.util` koja za smještanje i indeksiranje elemenata koristi hash tabelu.
- Vezano za `HashSet`, imamo pojmove **početnog kapaciteta** i **faktora opterećenja** (eng. *load factor*).
- Početni kapacitet je početna veličina niza vrijednosti (broj bucket-a), dok faktor opterećenja predstavlja procenat popunjenosti niza kad dolazi do dupliranja kapaciteta. Ove se vrijednosti zadaju preko konstruktora.
- Podrazumijevane vrijednosti početnog kapaciteta i faktora opterećenja su 16 i 0.75, respektivno, što znači da kada broj `HashSet` elemenata postane 12 = 0.75 × 16, kapacitet će se povećati na 32.

Metoda	Opis
`add`	Dodaje element (parameter metode) u `HashSet` ukoliko taj element ne postoji.
`addAll`	Nadovezuje elemente niza (drugi parametar) na kolekciju (prvi parametar)
`remove`	Uklanja objekat (parametar metode) iz skupa.
`clear`	Briše sve elemente iz skupa.
`contains`	Provjerava da li postoji element (parameter metode) u skupu i vraća `true` ako postoji, odnosno `false` ako ne postoji.
`isEmpty`	Vraća `true` ako je skup prazan, `false` u suprotnom.
`size`	Vraća broj elemenata skupa.

Klasa TreeSet

- `TreeSet` koristi stablo za smještanje elemenata.
- U `TreeSet`-u, za razliku od `HashSet`-a, elementi su sortirani. Prilikom dodavanja elemenata se vrši sortiranje, tj. element se smješta u stablo tako da bude zadovoljena sortiranost.
- Duplikati elemenata se eliminišu prilikom pokušaja dodavanja.
- Za realizaciju `TreeSet`-a, koriste se **samobalansirajuća binarna stabla pretrage** (eng. *self-balancing binary search tree*).
- Binarno stablo pretrage je binarno stablo kod koga za svaki čvor važi da je veći od svih čvorova u njegovom lijevom podstablu i manji od svih čvorova u njegovom desnom podstablu. Binarno stablo je balansirano ako se visina lijevog i desnog podstabla svakog čvora razlikuju maksimalno za 1. Binarno stablo je samobalansirajuće ako automatski ostaje balansirano nakon operacija unošenja i brisanja čvorova.



Samobalansirajuće binarno stablo pretrage

Klasa TreeSet

- `TreeSet` implementira **`SortedSet`** interfejs, koji nasljeđuje `Set` interfejs.
- Sve kolekcije koje implementiraju `SortedSet` interfejs imaju sortirane elemente, bilo u prirodnom poretku (npr. rastući redosljed za brojeve) ili u poretku koji definiše `Comparator` objekat proslijeđen `TreeSet` konstruktoru.
- Osnovne operacije sa elementima (dodavanje, uklanjanje i provjera da li element postoji) kod `TreeSet`-a su sporije nego kod `HashSet`-a, ali još uvijek mnogo brže nego kod nizova ili povezanih listi.
- Ako `TreeSet` sadrži N elemenata, potrebno je u prosjeku $\log_2 N$ poređenja za ove osnovne operacije, dok je kod `HashSet`-a složenost ovih operacija konstantna, tj. ne zavisi od broja elemenata.
- `TreeSet` ima četiri konstruktora. Pomenimo 1) konstruktor sa parametrom kolekcijom čije elemente kopira u kreiranu `TreeSet` instancu i sortira prema prirodnom poretku, i 2) konstruktor sa parametrom komparatorom koji definiše poredak elemenata.

Primjer sa Set-ovima

```
import java.util.Comparator;

public class KomparatorStringova implements Comparator<String> {
   public int compare(String prvi, String drugi) {
      return prvi.length() - drugi.length();
   }
}
```

Dužina stringa kao kriterijum poređenja

```
import java.util.*;

public class HashSetTreeSet {

   public static void main(String[] args) {
     String nizImena[] = {"Ana", "Marko", "Robert", "Esma", "Dalibor",
                          "Igor", "Ivana", "Marko", "Robert", "Ana"};

     Set< String > hSet = new HashSet< String >(Arrays.asList(nizImena));
     Set< String > tSet1 = new TreeSet< String >(hSet);
     Set< String > tSet2 = new TreeSet< String >(new KomparatorStringova());
     tSet2.addAll(tSet1);

     System.out.printf("HashSet:\n%s", hSet);
     System.out.printf("\nTreeSet (prirodni poredak):\n%s", tSet1);
     System.out.println("\nTreeSet (korisnički definisan poredak):");
     for(String ime: tSet2)
          System.out.print(ime + " ");
   }
}
```

```
HashSet:
[Esma, Igor, Ana, Robert, Marko, Dalibor, Ivana]
TreeSet (prirodni poredak):
[Ana, Dalibor, Esma, Igor, Ivana, Marko, Robert]
TreeSet (korisnički definisan poredak):
Ana Esma Ivana Robert Dalibor
```

Kad je dužina kriterijum poređenja stringova, isti su oni koji su iste dužine.

Mape

- Za razliku od skupova, koji sadrže samo vrijednosti, **mape** (eng. *map*) sadrže parove ključ-vrijednost (eng. *key-value*).
- Mapa je objekat koja mapira (preslikava) ključeve u vrijednosti. Ključevi u mapi moraju biti jedinstveni, dok odgovarajuće vrijednosti ne moraju biti. Ključevi i vrijednosti mogu biti proizvoljni referencijski tipovi.

Metoda	Opis
`put`	Dodaje ključ (prvi parametar) i odgovarajuću vrijednost (drugi parametar) u mapu. Ukoliko ključ već ima pridruženu vrijednost, metoda mijenja staru vrijednost novom. **Primjer**: Ako je ključ tipa `String`, a vrijednost `Integer`, naredba `mapa.put("Cetinje", 81250)` će dodati ključ `"Valjevo"` sa pripadajućom vrijednošću `81250` u mapu.
`putAll`	Kopira sve parove ključ-vrijednost iz mape (parametar metode) u mapu koja poziva.
`get`	Vraća vrijednost koja odgovara ključu (parametar metode), odnosno null ako ne postoji ta vrijednost. **Primjer**: Naredba `mapa.get("Cetinje")` će vratiti vrijednost `81250`, nakon naredbe iz primjera `put` metode.
`remove`	Uklanja par ključ-vrijednost iz mape za proslijeđeni ključ (parametar metode) iz mape. Metoda vraća vrijednost dodijeljenu ključu, odnosno `null` ako postoji predmeti par ključ-vrijednost.
`clear`	Briše sve parove ključ-vrijednost iz mape.
`containsKey`	Vraća `true` ako mapa sadrži mapiranje sa predmetnim ključem, `false` ako ne sadrži.
`containsValue`	Vraća `true` ako mapa sadrži jedno ili više mapiranja sa predmetnom vrijednošću, `false` ako ne sadrži.
`keySet`	Vraća `Set` izgled (eng. *Set view*) ključeva mape. Mapa podržava skup ključeva, tako da se promjene na mapi odražavaju na skup i obrnuto.
`values`	Vraća `Collection` izgled (eng. *Collection view*) vrijednosti sadržanih u mapi. Mapa podržava kolekciju, tako da se promjene na mapi odražavaju na kolekciju i obrnuto.
`entrySet`	Vraća `Set` izgled parova ključ-vrijednost sadržanih u mapi. Mapa podržava skup ključeva, tako da se promjene na mapi odražavaju na skup i obrnuto.
`isEmpty`	Vraća `true` ako je mapa prazna, `false` u suprotnom.
`size`	Vraća broj parova ključ-vrijednost mape.

Mape

- Neke od klasa koje implementiraju interfejs `Map` su **`HashTable`**, **`HashMap`** i **`TreeMap`**. `HashTable` i `HashMap` smještaju elemente u hash tabelama, dok ih `TreeMap` smješta u stablo.
- Interfejs **`SortedMap`** nasljeđuje `Map`, i klase koje implementiraju `SortedMap` imaju sortirane ključeve. Klasa `TreeMap` implementira `SortedMap`.
- `HashTable`, `HashMap` i `TreeMap` su generičke klase čije instance se kreiraju na sljedeći način (dajemo primjer sa `HashMap`):

```
Map<K,V> map = new HashMap<K,V>();
```

gdje `K` i `V` predstavljaju tip ključa odnosno vrijednosti.

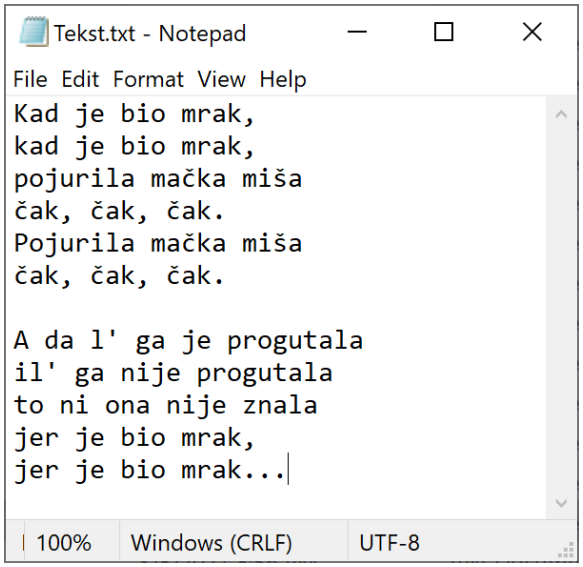
- Metode prikazane u tabeli sa prethodnog slajda možemo koristiti sa svakom od ove tri klase.
- U nastavku dajemo primjer korišćenja mape za brojanje pojava svake riječi u datom tekstualnom fajlu.

Primjer sa TreeMap

```
import java.io.IOException;
import java.nio.charset.StandardCharsets;
import java.nio.file.*;
import java.util.*;

public class RadSaMapom {

  public static void main(String[] args) {
    List< String > listaLinija = null;

    try{
      Path putanja = Paths.get("C:\\Temp\\Tekst.txt");
      listaLinija = Files.readAllLines(putanja, StandardCharsets.UTF_8); }
    catch (IOException e){
      System.err.println( "Greška prilikom čitanja fajla." );
      System.exit(1); }

    Map< String, Integer > mapa = new TreeMap< String, Integer >();

    for(String linija: listaLinija){
      linija = linija.toLowerCase();
      String rijeci[] = linija.split("[ ,.;!?-]");
      for(String rijec: rijeci){
        if(rijec.length() > 0)
          if(mapa.containsKey(rijec))
            mapa.put(rijec, mapa.get(rijec) + 1);
          else
            mapa.put(rijec, 1);
      }
    }
    Set< String > kljucevi = mapa.keySet();
    int k = 1;
    for(String rijec: kljucevi)
      System.out.printf("%-13s%3s%s", rijec,
       mapa.get(rijec), (k++ % 4 == 0)? "\n" : " | ");
  }
}
```

Primjer sa TreeMap

Fajl



Ispis

```
a          1 | bio        4 | da         1 | ga         2
il'        1 | je         5 | jer        2 | kad        2
l'         1 | mačka      2 | miša       2 | mrak       4
ni         1 | nije       2 | ona        1 | pojurila   2
progutala  2 | to         1 | znala      1 | čak        6
```

Komentari primjera

- Metoda **`readAllLines`** (klasa **`Files`** iz paketa `java.nio.file.Files`) čita sve linije fajla i vraća listu stringova, pri čemu jedna linija predstavlja jedan string.
- Metoda `readAllLines` zatvara fajl nakon čitanja linija. U suprotnom, metoda baca `IOException`.
- Argumenti metode `readAllLines` su:
  - putanja do fajla, koja uključuje i ime fajla (objekat klase **`Path`** iz paketa `ava.nio.file.Path`), i
  - skup karaktera na osnovu kojeg se kodiraju karakteri nakon čitanja (klasa **`StandardCharsets`** iz `java.nio.charset.StandardCharsets`).
- Pošto metoda `readAllLines` baca `IOExcetion`, koji je provjereni izuzetak, moramo obraditi taj izuzetak (`try` naredba) u metodi `main` ili deklarisati metodu `main` tako da baca izuzetak.
- Metoda **`exit(1)`** prekida trenutno izvršavanje JVM-a. Cjelobrojni argument je status kôd, čija nenulta vrijednost označava neregularan prekid.

Komentari primjera

- Metoda koja vrši upis linija u fajl je **`write`** (klasa **`Files`** iz paketa `java.nio.file.Files`) i ona se koristi na sljedeći način:

```
Files.write(putanja, listaLinija, StandardCharsets.UTF_8);
```

- Lista linija se navodi kao drugi argument.
- Naredba

```
Map< String, Integer > mapa = new TreeMap< String, Integer >();
```

kreira novu praznu `TreeMap`-u, pri čemu će se prilikom ubacivanja elemenata koristiti prirodan redosljed (npr. rastući za brojeve).

- Spoljašnja `for` petlja prolazi kroz sve stringove liste `listaLinija`, pri čemu jedan string predstavlja jedan red fajla. Nakon prebacivanja svih slova u mala, vršimo „razbijanje“ stringa `linija` na podstringove naredbom

```
String reci[] = linija.split("[ ,.;!?]");
```

gdje su u zagradi dati karakteri delimiteri. Ako želimo da imamo više delimitera, navodimo ih u uglastoj zagradi `[]`.

Komentari primjera

- Metoda **`containsKey`** vraća `true` ako tekuća riječ (ključ) postoji u mapi.
- Metoda **`put`** dodjeljuje vrijednost datom specifiranom ključu. Ukoliko dati ključ već ima vrijednost, metoda `put` mijenja staru vrijednost novom.
- Metoda **`get`** vraća vrijednost koja odgovara ključu, parametru metode, ili vrijednost `null` ako ključ ne postoji u mapi.
- Metoda **`keySet`** vraća `Set` izgled ključeva sadržanih u mapi. Dobijena `Set` kolekcija je podržana mapom, tako da se promjene na mapi odražavaju na kolekciju i obrnuto.
- Objašnjenje ovih i drugih metoda interfejsa `Map` je dato na 18. slajdu.

Prolazak kroz mapu

- Alternativni način prolaska kroz mapu je pomoću **`Map.Entry`** interfejsa.
- Štampanje svih elemenata mape se moglo odraditi na sljedeći način:

```
Set<Map.Entry<String, Integer>> set = mapa.entrySet();
int k = 1;
for(Map.Entry<String, Integer> element: set)
System.out.printf("%-12s%3d%s", element.getKey(),
                  element.getValue(), (k++ % 5 == 0) ? "\n" : " | ");
```

- `Map.entry` predstavlja par ključ-vrijednost.
- Metoda **`entrySet`** vraća kolekcioni izgled (*`Set view`*) mape (preslikavanja sadržanih u mapi).
- Metode **`getKey`** i **`getValue`** vraćaju ključ i vrijednost koji odgovaraju tekućem elementu mape.
- Postoji i metoda **`setValue`** koja postavlja novu vrijednost elementa mape.